

ARBEITSBERICHTE DES INSTITUTS FÜR MATHEMATISCHE MASCHINEN UND DATENVERARBEITUNG (INFORMATIK)

FRIEDRICH ALEXANDER UNIVERSITÄT ERLANGEN NÜRNBERG

**HERAUSGEBER: H. Billing, W. Händler, U. Herzog,
F. Hofmann, K. Leeb, P. Mertens, H. Müller, G. Nees,
H. Niemann, D. Seitzer, H. J. Schneider**

**Beschreibung von Programmmoduln
zum Entwurf zuverlässiger Softwaresysteme**

Werner Altmann

Dissertation

BAND 11 • NUMMER 16 • ERLANGEN • OKTOBER 1978

1978 #303



Digitized by the Internet Archive
in 2026

ARBEITSBERICHTE **DES INSTITUTS FÜR MATHEMATISCHE** **MASCHINEN UND DATENVERARBEITUNG** **(INFORMATIK)**

FRIEDRICH ALEXANDER UNIVERSITÄT ERLANGEN NÜRNBERG

HERAUSGEBER : H. Billing, W. Händler, U. Herzog,
F. Hofmann, K. Leeb, P. Mertens, H. Müller, G. Nees,
H. Niemann, D. Seitzer, H. J. Schneider

Beschreibung von Programmmoduln **zum Entwurf zuverlässiger Softwaresysteme**

Dem Fachbereich Ingenieurwissenschaften
der Universität Erlangen-Nürnberg
zur Erlangung des Grades
Doktor - Ingenieur

vorgelegt
von
Werner Altmann

Dissertation

BAND 11 · NUMMER 16 · ERLANGEN · OKTOBER 1978

ARBEITSBERICHT
DES INSTITUTS FÜR MATHEMATISCHE
MASCHINEN UND DATENVERARBEITUNG
(INFORMATIK)

FRÜHERE ALLEMANNO UNIVERSITÄT ERLANGEN-NÜRNBERG
VERGLEICHENDE: W. Seitzer, W. Hofmann, H. J. Schneider
F. Hofmann, K. Leeb, F. Hofmann, H. J. Schneider, G. Weier
H. Hofmann, G. Seitzer, H. J. Schneider

Bestimmung von Programmodul
zum Entwurf zuverlässiger Softwaresysteme

Als Dissertation genehmigt vom
Fachbereich Ingenieurwissenschaften
der Universität Erlangen-Nürnberg



Tag der Einreichung: 13. 7. 78
Tag der Promotion: 13. 10. 78
Dekan: Prof. Dr. D. Seitzer
Berichterstatter: Prof. Dr. H. J. Schneider
Prof. Dr. F. Hofmann

Werner Altmann

Dissertation

ZUSAMMENFASSUNG

Es wird ein Modul-Konzept entwickelt, das von der Einteilung eines Programmsystems in funktionale (den Kontrollfluß steuernde) Einheiten und solchen, die zur Definition komplexer Datenstrukturen dienen, ausgeht. Ein zentraler Punkt hierbei ist die Vorstellung, daß über Modulgrenzen hinweg nur geschützte Daten, in Form von Zugriffsrechten, weitergereicht werden können. Die formale Beschreibung der zwischen den Modulen zulässigen bzw. gewünschten Vernetzungen wird anhand eines graphischen Modells vorgenommen. Die Einbettung des Modul-Konzeptes in eine programmiersprachliche Umgebung sowie die Möglichkeiten zur Realisierung der Module in ALGOL-ähnlichen Sprachen, FORTRAN und Assemblern werden diskutiert. Eine Gegenüberstellung dieses Ansatzes mit einigen in der Literatur bekannten wird am Ende der Arbeit gegeben.

ABSTRACT

A module concept is developed which is based on the decomposition of programming systems into functional units (for control flow aspects) and those for the definition of complex datastructures. A central point thereby is the rule that through module interfaces only protected and encapsulated data can be passed via access rights. The formal description of permissible and desirable connections between modules is given by a graph model. The embedding of this module concept into a programming language environment as well as realization possibilities in ALGOL-like languages, FORTRAN and assemblers are discussed. Finally a comparison of this approach with some, known in the literature is given.

Motivation und Überblick	1
1 Darstellung eines Modul-Konzeptes	5
1.1 Ausgangspunkt	5
1.2 Begriffsbildung	7
1.3 Prinzipien des Systemaufbaus	10
1.4 Modulare Programmentwicklung	14
1.5 Zusatzbemerkungen	18
2 Formale Beschreibung modularer Programm- systeme	21
2.1 Vorbemerkungen	21
2.2 Graphmodell eines modularen Programm- systems (Modulgeflecht)	24
2.3 Darstellung von Modulgeflechten	52
2.4 Anwendungsbeispiele	59
3 Entwurf einer programmiersprachlichen Notation	73
3.1 Begriffsbildung	74
3.2 Sprachkonstruktionen im Zusammenhang mit D-Moduln	82
3.3 Sprachkonstruktionen im Zusammenhang mit F-Moduln	90
3.4 Beispiele	97

	Seite
4 Implementierungsaspekte	105
4.1 Zielsetzung	105
4.2 Beschreibung von Problemen zur Überset- zungszeit	106
4.3 Beschreibung von Laufzeitaspekten	131
5 Modulare Programmierung in blockorientierten Sprachen, FORTRAN und Assemblern	140
5.1 Zielsetzung, Vorbemerkungen	140
5.2 Realisierung von D-Moduln	142
5.3 Realisierung von F-Moduln	153
6 Grenzen des Konzeptes	160
7 Vergleiche mit Ansätzen in der Literatur . . .	171
Schlußbemerkung	176
Stichwortverzeichnis	177
Literaturverzeichnis	179

MOTIVATION UND ÜBERBLICK

In den fünfziger Jahren wurden höhere Programmiersprachen wie FORTRAN und ALGOL 60 entwickelt, um für ein gegebenes Problem maschinenunabhängige und möglichst problemnahe Lösungen und Algorithmen angeben zu können. Um die Verwendbarkeit dieser beiden Sprachen für das Lösen von Problemen besser einschätzen zu können, sei kurz erläutert, wie man für eine gegebene Problemstellung einen Lösungsalgorithmus entwickeln kann.

Ausgehend von einer intuitiv vorhandenen Lösungsidee eines Problems kann zunächst eine abstrakte Darstellung des Algorithmus in irgendeiner Form (z.B. Bilder mit verbalen Erläuterungen) gefunden werden. Diese abstrakte Lösung kann nun unter Zuhilfenahme einer Programmiersprache in eine algorithmische Form gebracht werden. Die so entstandene Darstellung des Lösungsalgorithmus muß dann in eine von einem Compiler übersetzbare Form codiert werden, womit die Arbeit des Programmierers beendet ist.

Unter anderem kann man sich bei der Entwicklung einer Lösung der folgenden Hilfsmittel zur Darstellung bedienen:

- funktionale (prozedurale) Abstraktion

Von der Kontrollstruktur einer Teillösung wird durch Verwendung einer Funktion oder Prozedur abstrahiert.

- datenorientierte Abstraktion

Die Bearbeitung von Daten in einem Programm macht keinen Gebrauch von deren strukturellem Aufbau. Die Schnittstelle zu den Daten ist gegeben durch die auf ihnen definierten Verarbeitungsoperationen.

- syntaktische Abstraktion

Abweichend von der Notation einer vorgegebenen Zielsprache können zur Formulierung eines Algorithmus syntaktische Konstruktionen für neue Kontrollstrukturen definiert, d.h. die Syntax der Sprache "erweitert" werden (vgl. z.B. [ALT 74]).

Sehr frühzeitig wurde offenbar die Notwendigkeit der prozeduralen Abstraktion in Programmiersprachen erkannt, da bereits die "Erstlinge" FORTRAN und ALGOL 60 sehr brauchbare Prozedurkonzepte enthalten. Die Möglichkeiten, komplexere Datenstrukturen zu definieren, beschränken sich bei beiden Sprachen auf die Verwendung von ganzzahligen und reellwertigen Feldern. Mechanismen zur Erweiterung der Syntax fehlen ganz.

Ende der sechziger, Anfang der siebziger Jahre wurde eine Reihe von Vorschlägen veröffentlicht, die sich mit Mechanismen zur syntaktischen bzw. datenorientierten Erweiterung von Programmiersprachen befassen, z.B. [LEA 66], [SCJ 70], [DIC 71] und [ALT 74]. Eine ausführliche Übersicht zum Thema "Erweiterbare Programmiersprachen" findet man bei [SOY 74].

Während sich die Ansätze bzgl. der syntaktischen Erweiterbarkeit nicht durchgesetzt haben (vgl. etwa [STA 75]), fanden neuere Konzepte zur Strukturierung und Definition komplexer Datenstrukturen ihren Niederschlag in vielen neuentwickelten Programmiersprachen wie SIMULA 67 [DMN 70], ALGOL 68 [WIJ 69], PASCAL [WIR 71] und MODULA [WIR 77]. Der wohl vielversprechendste Weg dürfte dabei in SIMULA 67 beschritten worden sein, da das class-Konzept in SIMULA 67 als der Ausgangspunkt der gegenwärtigen Überlegungen - im Zusammenhang mit abstrakten Datentypen - gesehen werden muß.

Daß syntaktische Erweiterbarkeit nicht unbedingt nötig und in vielen Fällen auch gar nicht wünschenswert ist, hat die Ende der sechziger, Anfang der siebziger Jahre geführte Diskussion um die Methode der "Strukturierten Programmierung" [DDH 72] und die damit verbundene "goto"-Kontroverse gezeigt. Zur Formulierung von Algorithmen genügt es, einige "wohldefinierte" Kontrollstrukturen, womit nicht nur die "klassischen" gemeint sind, zu verwenden.

Das starke Anwachsen der Kosten zur Erstellung von Software in den letzten Jahren stellt erhöhte Forderungen an moderne Programmiersprachen und an die zu verwendenden Programmiermethoden^{*)}. Nicht nur die hohen Kosten zur Programmerstellung, sondern auch die Größe der Programme waren dafür ausschlaggebend, daß man begann, nach einem Ausweg aus der "Software-Krise" zu suchen. Die Veröffentlichungen [BAK 72] und [BAM 73] über ein Programmierprojekt bei der New York Times und den dabei verwendeten Programmiermethoden sowie der speziellen Team-Organisation sind Ausdruck hierfür.

Unter anderem lassen sich folgende Anforderungen an eine moderne Programmiersprache bzw. -methode stellen:

- *Einfachheit* der Darstellung, d.h. keine komplizierten Konzepte
- *Hoher Dokumentationswert*, d.h. leichte Lesbarkeit des Codes
- *Unterstützung* bei der Erstellung zuverlässiger Software
- *Beweisbarkeit* von Systemeigenschaften
- *Änderungsfreundlichkeit* der Programme

^{*)} Eine vergleichende Darstellung der bekannten Programmiermethoden ist in [ALW 77] gegeben.

Die vorliegende Arbeit befaßt sich mit Problemstellungen beim Entwurf großer Programmsysteme. Eine Vielzahl der behandelten Aspekte ist daher nur in diesem Zusammenhang von Interesse und spielt bei der Erstellung kleiner Programme keine Rolle. Im folgenden werden ein Überblick und eine Gliederung der Arbeit gegeben.

Ausgehend von der Einteilung eines Programmsystems in funktionale (den Kontrollfluß steuernde) Einheiten und solche zur Definition komplexer Datenstrukturen werden in Kapitel 1 allgemeine Prinzipien zum Entwurf zuverlässiger Systeme dargestellt. Eine formale Beschreibung der Struktur von modularen Programmsystemen sowie der Interaktion zwischen Modulen wird in Kapitel 2 anhand eines graphischen Modells gegeben.

Die beiden darauffolgenden Kapitel beschreiben die Einbettung des vorgestellten Modul-Konzeptes in eine programmiersprachliche Umgebung und die Probleme, die sich für einen entsprechenden "Modul-Compiler" sowohl zur Übersetzungs- als auch für die Laufzeit ergeben.

Daß man auch in blockorientierten Sprachen, wie z.B. ALGOL 60, in FORTRAN und selbst in Assemblern, bei Einhaltung einer entsprechenden Disziplin modular programmieren kann, wird in Kapitel 5 gezeigt.

Anhand der Programmierung "mehrstelliger Operationen" auf komplexen Datenstrukturen und des "prison mail system"-Problems (vgl. [AHM 77]) sollen in Kapitel 6 die Grenzen des Konzeptes in bezug auf Effizienz und Eignung aufgezeigt werden.

Zum Schluß wird der hier gewählte Ansatz mit einigen in der Literatur bekannten verglichen.

KAPITEL 1 DARSTELLUNG EINES MODUL-KONZEPTES

1.1 AUSGANGSPUNKT

In der Literatur werden die Begriffe *modular*, *Modul*, *Modularität* und *Modularisierung* sehr unterschiedlich verwendet. Als eine der wenigen Gemeinsamkeiten findet man die Vorstellung, daß ein Modul eine abgeschlossene Hardware- oder Softwareeinheit darstellt.

Aber selbst bei einer Beschränkung auf den Bereich Software können für die Zielsetzungen der Modularisierung eines Systems (a) bzw. die den Modulen zugedachten Eigenschaften und Aufgaben (b) voneinander abweichende Konzepte festgestellt werden:

- a) Parnas [PAR 72b] sieht u.a. in der *Zerteilung* eines Systems in handhabbare Einheiten einen wichtigen Aspekt bei der Modularisierung. Bei der Entwurfsmethode HIPO [STA 76] steht eine *hierarchische Struktur* im Vordergrund. Wesentlich für die von Koster entwickelte Sprache CDL2 (vgl. etwa [HAE 76]) sind Möglichkeiten zur Definition von *Programmschichten*.
- b) Die Steuerung *konkurrierender* Zugriffe zu den in einem Betriebssystem verwalteten Daten ist der Ausgangspunkt für das monitor-Konzept von Hoare und Hansen ([HOA 74], [HAN 75]).
- c) Die Vorstellung, daß zusammengehörige Daten und alle darauf definierten Operationen einen Modul bilden, liegt einer Reihe von Arbeiten zum Thema "*abstrakte Datentypen*" zugrunde ([LIZ 74], [KOS 74], [WUL 74]).
- d) Verschiedentlich wird auch die Forderung nach *Kontext-unabhängigkeit* von Programmodulen gestellt ([DEN 72]).

Das hier vorgestellte Modulkonzept geht von einer *sequentiellen Programmierumgebung* aus, d.h. es werden Probleme, die sich bei der Steuerung und Synchronisation konkurrierender Prozesse ergeben, ausgeklammert. Ziel ist die Darstellung einer Modularisierungs-Methode, der gewisse Prinzipien zugrunde liegen. Ausgangspunkt der Überlegungen waren hierbei die Arbeiten von Parnas [PAR 71], [PAR 72a], [PAR 72b] und [PAS 75], eine Arbeit von De Remer/Kron [DEK 76] sowie die Arbeiten von Liskov und Zilles [ZIL 73], [LIZ 74] und [LIZ 75].

1.2 BEGRIFFSBILDUNG

Zur Abgrenzung gegenüber der in der Literatur verwendeten Terminologie werden in diesem Abschnitt "Arbeitsbegriffe" eingeführt.

Definition 1.1

Ein Programmbaustein heißt *Modul* oder *modular*, wenn er den folgenden Kriterien genügt:

Der Baustein

- läßt erkennen, daß er eine *Entwurfsentscheidung realisiert*,
- beinhaltet eine *vollständige Beschreibung der Annahmen gegenüber anderen Systembausteinen (Schnittstellenspezifikation^{*)}*,
- ermöglicht die *Formulierung von Schutzaspekten^{**}*,
- ist als *korrekt nachweisbar* allein unter Verwendung der Annahmen, die gegenüber anderen Bausteinen gemacht werden (d.h. insbesondere ohne Kenntnis über deren internen Aufbau),
- ist unabhängig von seiner Umgebung, d.h. *kontextunabhängig^{***}*.

□

^{*)} Der Begriff "Schnittstellenspezifikation" umfaßt nur die Beschreibung des durch die Parameter definierten E/A-Verhaltens und ist zu unterscheiden von dem Begriff "Spezifikation einer Datenstruktur" bzw. eines "Datentyps", wie er etwa in [WUL 76] oder [GUT 75] verwendet wird.

^{**)} d.h., es muß möglich sein, Information über den Baustein vor unerlaubtem Zugriff von außen zu schützen und abhängige (z.B. aufgerufene) Programmteile daran zu hindern, auf nicht für sie bestimmte Daten zuzugreifen.

^{***)} Dies bedeutet u.a., daß ein Modul keine globalen Größen verwendet und getrennt übersetzbar ist.

Definition 1.2

Unter *Modularer Programmierung* (MP) versteht man eine Methode zur Erstellung von Programmen und Programmsystemen, wobei die Grundlage für Entwurfsentscheidungen und Zerlegungen Bausteine sind, deren Aufbau modular ist.

□

MP ist somit unabhängig von den Begriffen "top down"-[DDH 72], "bottom up"- und "outside in"-Entwicklung [PAS 75].

Im folgenden wird eine Charakterisierung von "Modul-Arten" angegeben.

Ein Programmsystem kann aufgeteilt werden in logisch abgeschlossene Einheiten zur Definition des Kontrollflusses, d.h. des dynamischen Ablaufs, bzw. zur Schaffung von Informationsbehältern, die zu jedem Zeitpunkt Auskunft über den Systemzustand geben. Diese Unterscheidung legt folgende Begriffsbildung nahe:

Definition 1.3

Unter einem *funktionalen Modul* (abgek. *F-Modul*) versteht man einen Programmbaustein, der zur Darstellung des dynamischen Programmablaufs dient und modular aufgebaut ist.

Unter einem *Datenmodul* (abgek. *D-Modul*) versteht man einen Programmbaustein, der modular aufgebaut ist und zur Darstellung von Information dient, deren Manipulation in eindeutiger Weise durch eine Menge von zugeordneten *Zugriffsoperationen* definiert ist. Ein D-Modul beinhaltet somit die Definition einer Datenstruktur (im folgenden auch Repräsentierung genannt), einschließlich aller darauf erklärten Verarbeitungs(Zugriffs-)operationen.

□

Die Zugriffsoperationen eines D-Moduls können *lesender*,

schreibender oder gemischter Natur sein.

F-Moduln sind im Gegensatz zu D-Moduln *gedächtnislos*, d.h. daß die innerhalb eines F-Moduls definierten (F-Modul-lokalen) Daten nur während der "Ausführungsdauer" eines Modul-Aufrufs verfügbar sind.

Bezüglich der lokalen Daten verhält sich ein F-Modul somit wie ein "Block" einer ALGOL-ähnlichen Programmiersprache.

Definition 1.4

Ein *Zugriffsrecht* ist ein Paar (d, z) , wobei d ein D-Modul und z eine durch d definierte Zugriffsoperation ist.

□

Durch einen D-Modul wird - zusammen mit den darauf definierten Zugriffsoperationen - eine *Menge von Zugriffsrechten* erklärt.

Mit Hilfe der Moduln kann nun ein *modulares Programmsystem* aufgebaut werden. Es wird beschrieben durch^{*)}

- eine Menge von F- und D-Moduln,
- Verbindungen der F- und D-Moduln untereinander, (wobei nicht beliebige Vernetzungen zugelassen sind; vgl. Kap. 1.3)
- eine Darstellung der Einbettung des Systems in die jeweilige Programmierungsumgebung^{**) .}

^{*)} Eine formale Beschreibung wird in Kap. 2 gegeben.

^{**)} Die Programmierungsumgebung besteht aus vorgefertigten Programmteilen (Bibliotheksprogrammen), z.B. Hilfs- und Fehlerrountinen.

1.3 PRINZIPIEN DES SYSTEMAUFBAUS

Neben den in Kap. 1.2, Def. 1.1 angegebenen Modul-Eigenschaften verlangt man für die zwischen den Moduln zulässigen Verbindungen zum Aufbau eines Gesamtsystems zusätzliche Eigenschaften.

Für diese Zusatzbedingungen werden im folgenden konzeptuelle Begriffe eingeführt, deren formale Definition in Kapitel 2 an Hand des dort eingeführten Graphmodells erfolgt.

- Die Menge der F- und D-Moduln sind jeweils *hierarchisch geordnet* (vgl. Def. 2.2, 2.3 und 2.4).
- Die *Verteilung* der durch die D-Moduln definierten *Zugriffsrechte* ist *verträglich mit der Hierarchie* der F- und D-Moduln. Dies bedeutet im wesentlichen, daß übergeordnete Moduln bezüglich eines D-Moduls stets mehr Zugriffsrechte besitzen als untergeordnete Moduln (vgl. Def. 2.5 und 2.6).
- Der Aufbau des Systems ist *verträglich mit dem Prinzip der zentralen Ressourcen-Verwaltung*. Dies bedeutet, daß zwischen je zwei (gemäß der Hierarchie) untergeordneten Moduln ein Informationsaustausch nur unter der Kontrolle des (direkt) übergeordneten stattfinden kann (vgl. Satz 2.3).
- Die zwischen Moduln zulässigen Verbindungen genügen den Prinzipien "*information hiding*" bzw. "*information security*" (vgl. [PAR 71]) und "*uniform reference*" (vgl. [GEM 75]). Im allgemeinen bedeutet dies, daß ein Aufruf eines Moduls von dessen Internstruktur keinen Gebrauch machen darf ("*information hiding*"). Im Fall der D-Moduln bedeutet "*uniform reference*", daß die Verwendung der Zugriffsoperationen von der Repräsentierung der Daten unabhängig ist (vgl. auch Def. 2.7).

Unter Berücksichtigung der hierarchischen Systemstruktur lassen sich prinzipiell zwei verschiedene Systemarten unterscheiden.

Man kann von einem *funktionsorientierten System* sprechen, wenn die Spitze der Hierarchie durch einen F-Modul gegeben ist. Die Funktion dieses F-Moduls bestimmt dann die Systemfunktion. Somit gibt es genau einen Eingang in das System (vgl. Def. 2.10).

Beispiele hierfür sind etwa ein Compiler und das in Kap. 2.4 angegebene System zur Steuerung einer Produktionsanlage.

Wird die Spitze der Modulhierarchie von einem D-Modul gebildet, so kann man von einem *datenorientierten System* sprechen. Die verschiedenen D-Modul-Operationen können dann als die Eingänge in das System betrachtet werden (vgl. Def. 2.10). Beispiele hierfür sind etwa Datenbanksysteme, Betriebssysteme und das in Kap. 2.4 angegebene Speichersystem DATAS.

Aus den in den Definitionen 1.1 und 1.3 angegebenen Eigenschaften und Charakterisierungen von Moduln läßt sich jeweils für F- bzw. D-Moduln ein prinzipieller Aufbau ableiten.

Definition 1.5

Ein *F-Modul* ist definiert durch folgende Komponenten:

- Darstellung der *Schnittstelle nach außen*.

Hierbei wird festgelegt, welche Zugriffsrechte dem F-Modul von der aufrufenden Umgebung übertragen werden müssen (Zugriffe zu *globalen* D-Moduln).

- Darstellung der *Schnittstelle nach innen*.

Die Schnittstelle nach innen setzt sich zusammen aus

- den innerhalb des F-Moduls definierten (*lokalen*) D-Moduln und den damit verbundenen Zugriffsrechten,
 - einer *Spezifikation der untergeordneten F-Moduln*, zusammen mit den dabei definierten "Informationskanälen"*) ,
 - einer Beschreibung der *Einbettung in die Programmierungsumgebung* (Externbezüge zu Hilfsroutinen).
- Formulierung des die Funktion des F-Moduls realisierenden *Algorithmus*.

□

Definition 1.6

Ein *D-Modul* ist definiert durch folgende Komponenten:

- Darstellung der *Schnittstelle nach außen*.

Sie zerfällt bei D-Moduln in zwei Anteile:

- *Zur Verfügung gestellte Schnittstelle nach außen*.

Hierbei werden die auf dem D-Modul definierten Zugriffsooperationen und die Spezifikation des E/A-Verhaltens dieser Operationen angegeben.

- *Von außen übertragene Schnittstelle*.

Sie beinhaltet die Kreierungsparameter**) für die interne Datenstruktur und eine Beschreibung der von der Aufrufumgebung übertragenen Zugriffsrechte (analog zu F-Moduln).

*) Zwischen zwei F-Moduln besteht ein Informationskanal, wenn sie Zugriffe bzgl. des gleichen D-Moduls besitzen. Die Darstellung der Informationskanäle soll dabei alle dynmaisch möglichen Bezüge von untergeordneten F-Moduln zu bereitgestellten D-Moduln sichtbar machen.

**) Kreierungsparameter sind Ladezeitkonstanten, die den Aufbau und Umfang der Datenstruktur eines D-Moduls definieren.

- *Darstellung der Schnittstelle nach innen.*

Sie umfaßt die innerhalb eines D-Moduls spezifizierten F- bzw. definierten D-Moduln sowie deren Ver- netzung.

- *Definition der Repräsentierung* (unter Verwendung der D-Moduln)
- *Realisierung der Zugriffsoperationen* in Abhängigkeit der Repräsentierung.
- *Beschreibung der Einbettung in die Programmierungsumgebung.*

□

Betrachtet man die Schnittstellen der Moduln nach außen, so erkennt man, daß - abgesehen von den Kreierungsparametern bei D-Moduln, für die man i.a. jede beliebige Konstante zulassen kann - über Modulgrenzen hinweg nur geschützte (d.h. in D-Moduln "verpackte") Daten weiterge- reicht werden können.

1.4 MODULARE PROGRAMMENTWICKLUNG

Als Grund für die Modularisierung eines Problems wird häufig angegeben, daß "große Programme" nur unter Einsatz mehrerer "Mannjahre" zu bewältigen und eben deshalb zu zerteilen (zu modularisieren) sind. Sehr viel wichtiger ist es, seine Probleme und Überlegungen derart aufzuteilen, daß der Gesamtaufwand an Gedankenarbeit, und damit auch für Codierung und Testen reduziert wird (vgl. auch [DIJ 76]).

Auf die Frage "Wie erhält man ein modular aufgebautes Programmsystem?" lassen sich keine allgemein gültigen Vorschriften angeben. Vor der Festlegung der Modulstruktur muß man sich jedoch Klarheit über folgende Punkte verschaffen:

- Auswirkungen der *Reihenfolge von Entwurfsentscheidungen*
- *Kontrolle der Informationsverteilung*
- *Kriterien der Zerlegung*
- *Richtung des Zerlegungsprozesses.*

Bei der Festlegung der *Reihenfolge von Entwurfsentscheidungen* lassen sich nach [PAR 71] die folgenden drei Ziele verfolgen:

- Das System soll *gute Eigenschaften nach außen hin haben, d.h. benutzerfreundlich sein.*
Dazu muß die Benutzerschnittstelle frühzeitig und vollständig festgelegt werden.
- Das System soll *möglichst schnell fertiggestellt werden.*
Dies kann im allgemeinen nur dann erreicht werden, wenn das System sehr frühzeitig zerteilt wird, damit viele Personen gleichzeitig an dem Projekt arbeiten können.

Bei dieser Art der Zerlegung, die u.U. ohne Kenntnis

einer vollständigen Benutzerschnittstelle vorgenommen wurde, kann es passieren, daß die entstandenen Teilaufgaben nicht den eigentlichen Systemkomponenten entsprechen.

- Das System soll *flexibel*, d.h. *leicht zu ändern* sein. Da Entscheidungen aufeinander aufbauen und somit voneinander abhängen, haben Änderungen der zuerst getroffenen die weitreichendsten Konsequenzen, während die letzten am leichtesten zu korrigieren sind. Um nun eine geringe Reichweite bzw. Wahrscheinlichkeit von Änderungen zu erreichen, müssen die ersten Entwurfsentscheidungen sehr sorgfältig und möglichst umgebungsunabhängig festgelegt werden.

Daß es überhaupt ein Entscheidungsproblem gibt, liegt nun daran, daß sich die aufgeführten Ziele teilweise widersprechen. In der Praxis wird man daher einen Kompromiß zwischen den verschiedenen Zielen suchen müssen.

Da Programmierer bei der Realisierung einer ihnen übertragenen Aufgabe im allgemeinen alle zur Verfügung gestellten Informationen ausnutzen, ist auf eine *kontrollierte Informationsverteilung* besonders Wert zu legen.

Je umfangreicher die verfügbare Information ist, desto stärker ist der von dem Programmierer erstellte Modul im Gesamtsystem verwurzelt^{*)}. Dies führt zu einer hohen Konnektivität desselben und verwischt nicht selten die vom Entwerfer intendierte Struktur. Die von mehreren Systemmoduln aus zugreifbare Information muß deshalb entsprechend gekapselt und gezielt verteilt werden (vgl. auch "zentrale Ressourcen-Verwaltung", Kap. 1.3).

Sehr häufig werden Programme entwickelt und in Teilpro-

^{*)} Die Austauschbarkeit des entsprechenden Programmbausteins wird dann durch unnötige Abhängigkeiten stark erschwert.

gramme zerlegt, indem man von einem Ablaufplan ausgeht, der das zeitliche Verhalten des Systems darstellt. Den einzelnen Schritten des Ablaufplanes wird dann ein Modul zugeordnet. Dies hat häufig zur Folge, daß wichtige Entwurfsentscheidungen (etwa die Speicherungsform von Daten) auf sehr viele auf diese Weise gewonnene Moduln verstreut werden. Änderungen solcher Entscheidungen sind daher mit hohem Aufwand verbunden.

Parnas schlägt in [PAR 72b] andere *Kriterien der Zerlegung* vor:

Um ein entflochtenes, und damit änderungsfreundliches, System zu erhalten, geht man bei der Zerlegung von einer Liste von Entwurfsentscheidungen aus. Den Entscheidungen wird nun jeweils ein Modul zugeordnet und derselbe so implementiert, daß er seine Realisierung nach außen verbirgt.

Zerlegungsrichtungen, bei denen von einer genauen Systemspezifikation ausgehend die interne Struktur abgeleitet wird, charakterisiert man mit den Begriffen "*top down*" oder "*outside in*".

Bei der Methode der "top down"-Entwicklung, die auf Dijkstra (vgl. [DDH 72]) zurückgeht, kann es passieren, daß Systemteile, die in der Hierarchie sehr weit "oben" liegen, im System sehr weit "unten" verankert sind, wenn die Entwurfsentscheidungen für deren Verfeinerung (zeitlich gesehen) sehr spät erfolgt sind. Die starke Verankerung im System ergibt sich nun aus der Tatsache, daß die sehr spät getroffenen Entscheidungen von sehr vielen (wenn nicht allen) vorangegangenen abhängen.

Anders ist dies bei der von Parnas und Siewiorek in [PAS 75] vorgestellten "outside in"-Entwicklungsmethode. Hier ist die Reihenfolge der Entwurfsentscheidungen Maß-

stab für die Hierarchisierung.

Bei den beiden zuletzt genannten Entwicklungsmethoden treten erhebliche Schwierigkeiten auf, wenn ein Programmsystem zu entwerfen ist, bei dem die Benutzerschnittstelle noch nicht klar festgelegt ist. Zur Umgehung dieser Probleme soll mit Hilfe des *Konzeptes der Transparenz* (vgl. [PAS 75]) eine "bottom up"-Entwurfsmethode vorgestellt werden.

Wird auf einer Basismaschine^{*)} eine Programmschicht implementiert, so entsteht eine *virtuelle Maschine*^{**)}. Eine solche Implementierung heißt *transparent*, wenn durch Programmierung der virtuellen Maschine alle (erwünschten) Möglichkeiten der Basismaschine erreichbar sind. Unter Umständen wird man zur Vermeidung *unerwünschter* Situationen (bei Betriebssystemen etwa den Zustand der Verklemmung) beim Übergang von der Basis- zur virtuellen Maschine einen *Verlust an Transparenz* erreichen wollen.

Diese "bottom up"-Methode kann zur Realisierung von Programm- und Systemfamilien herangezogen werden:

Es wird zunächst eine "genügend hohe" virtuelle Maschine (häufig als Grundsoftware bezeichnet) "bottom up" entworfen. Die einzelnen Systeme der Familie können dann mit Hilfe eines "outside in"-Entwurfs entwickelt werden.

^{*)} Hierunter kann man sich einen Rechner vorstellen, auf dem nur ein Assembler und der Kern eines Betriebssystems verfügbar ist.

^{**)} Die Befehle der virtuellen Maschine sind die Funktionen und Prozeduren der Programmschicht.

1.5 ZUSATZBEMERKUNGEN

Zum Abschluß dieses Kapitels seien zunächst noch einige Bemerkungen zur *Behandlung von Fehlerausgängen* gemacht. Im Anschluß daran wird kurz auf das Verhältnis der hier vorgestellten F-Moduln zu dem z.B. in ALGOL 68 vorhandenen *Block- bzw. Prozedurkonzept* eingegangen.

Bei der Bearbeitung von Daten treten im allgemeinen Zustände und Bedingungen auf, die eine *gezielte Reaktion durch das verarbeitende Programm* erforderlich machen. Eine zuverlässige Programmiermethode verlangt nun, daß relevante Ausnahmesituationen, die bei der Bearbeitung von Daten *innerhalb* von D-Moduln auftreten, auch *außerhalb* erkennbar sind. Die *Schnittstelle nach außen* eines D-Moduls muß deshalb auch entsprechende *Fehlerausgänge* bereitstellen. Weiter muß eine auf dem Modul-Konzept aufgesetzte Programmiersprache Anweisungsformen zur Verwendung der Fehlerausgänge beinhalten. Hierfür bieten sich PL/1-ähnliche ON-Bedingungen an. Die "Aktivierung" der entsprechenden Fehlerbehandlung durch das "Setzen" einer Bedingung geschieht durch die auf dem jeweiligen D-Modul definierten Zugriffsoperationen.

Bei der Übertragung von Zugriffsrechten an einen Modul ist zur Vermeidung von Nebenwirkungen darauf zu achten, daß alle Fehlerausgänge, die von einer einem Zugriffsrecht zugrunde liegenden Zugriffsoperation aktiviert werden können, mit Übertragen werden.

Damit ist es möglich, daß Fehler, die von einem Programmmodul durch den Aufruf einer Zugriffsoperation verursacht werden, auch in diesem Modul behandelt werden können.

Diese Forderung nach lokaler Fehlerbehandlung macht andererseits die Verwendung von ON-Bedingungen einer formalen (und kontextunabhängigen) Verifikation zugänglich (vgl. auch [GOO 75]).

Wenden wir uns nun dem Block- bzw. Prozedurkonzept zu. Ein wesentliches Kriterium der Modularität ist die Möglichkeit zur Formulierung von Schutzaspekten. Im Zusammenhang mit F-Moduln wird dabei von zentraler Ressourcen-Verwaltung und der Einschränkung von Zugriffsrechten auf D-Moduln bei untergeordneten F-Moduln gesprochen.

Die innerhalb eines *Blockes* definierten Daten sind zwar einerseits nur dort (d.h. in dem Gültigkeitsbereich des jeweiligen Blockes) zugreifbar, es kann jedoch andererseits nicht verhindert werden, daß die Zugriffsberechtigung zu diesen Daten an *alle* untergeordneten Blöcke automatisch vererbt wird. Ein Grund für diesen offensichtlichen Mangel des Blockkonzeptes ist das Fehlen einer explizit angebbaren Schnittstelle zwischen Blöcken.

Prozeduren erlauben durch die Parameterlisten die Angabe von Schnittstellen und ermöglichen dadurch eine kontextunabhängige Formulierung, sind aber aus nachfolgend geschilderten Gründen zur Verwendung als funktionale Moduln ungeeignet.

Die Kontextunabhängigkeit von Prozeduren führt im allgemeinen zu sehr langen Parameterlisten und zu einer Zunahme an globalen Variablen in übergeordneten Prozeduren, die dann als aktuelle Parameter der (kontextunabhängigen) Prozeduren auftreten. Die Häufung von globaler, zum Großteil voneinander unabhängiger Information in übergeordneten Prozeduren ist ebenso als nachteilig zu bezeichnen wie die Tatsache, daß eine korrekte Verwendung der globalen Daten nicht mehr an zentralen Stellen im System nachweisbar ist. Man muß vielmehr alle Programmbereiche, welche auf Teile der globalen Daten zugreifen, hinsichtlich ihres zugrunde liegenden Algorithmus untersuchen, um die korrekte Verwendung der Daten zu testen. Dieser Umstand steht im Widerspruch zum Prinzip "information

hiding". Die erreichte Kontextunabhängigkeit der Prozeduren steht im Widerspruch zur Kontextunabhängigkeit der Daten.

In einem gewissen Maß sind zwar Bedingungen, die an Prozedurparameter gestellt werden, durch entsprechende Testanweisungen innerhalb einer Prozedur formulierbar, die ausschließliche Bearbeitung der übergebenen Daten durch bestimmte Zugriffsoperationen (vgl. Modularitätskriterium "Schutzaspekt", Kapitel 1.2) kann jedoch nicht erzwungen werden.

KAPITEL 2 FORMALE BESCHREIBUNG MODULARER PROGRAMMSYSTEME

2.1 VORBEMERKUNGEN

Im nachfolgenden Kapitel werden die Begriffe *modulares Programmsystem*, *F-Modul* und *D-Modul* präzisiert und formal definiert. Der Ansatz ist gedacht zur Beschreibung von Vernetzungen in einem modularen Programmsystem. Die Darstellung eines solchen Systems als (*konsistentes*) *Modulgeflecht* stützt sich dabei auf den Begriff *Diagramm*, wie er in [SCH 70] definiert wurde. Des weiteren enthält das Kapitel Vorschläge zur Darstellung von Modulgeflechten in der Form von Schalt- bzw. Netzplänen.

Zunächst seien noch einige Begriffe und Notationen erläutert, die in den Definitionen dieses Kapitels verwendet werden.

I. Sei R eine Relation auf einer Basis (Knoten-)menge K , d.h.

$$R \subseteq K \times K.$$

Die Relation R^{-1} ist definiert durch:

$$R^{-1} = \{(k, k') \mid (k', k) \in R\}.$$

Die *transitive Hülle* R^+ der Relation R ist definiert durch:

$$\begin{aligned} (k, k') \in R^+ &: \Leftrightarrow \exists n \in \mathbb{N}, n > 0 \exists \{k_i\}_{i=1}^n, k_i \in K \forall 1 \leq i \leq n-1 \\ &((k_j, k_{j+1}) \in R \wedge k = k_1 \wedge k' = k_n). \end{aligned}$$

Die Menge der *maximalen Elemente* von R (in Zeichen: $\max(R)$) ist definiert durch:

$$\max(R) = \{k \mid \neg \exists k' \in K ((k', k) \in R)\}.$$

R besitzt *Waldstruktur*, wenn gilt:

$$\forall k_1, k_2, k \in K ((k_1, k) \in R \wedge (k_2, k) \in R \Rightarrow k_1 \neq k \wedge k_2 \neq k \wedge k_1 \neq k_2).$$

R besitzt *Baumstruktur*, wenn gilt:

$$R \text{ besitzt Waldstruktur} \wedge$$

$$|\max(R)| = 1$$

Durch eine nichtleere endliche Folge $\{k_i\}_{i=1}^n, k_i \in K$, wird ein *ungerichteter Kantenzug* (bzgl. R) definiert, wenn gilt:

$$\forall 1 \leq i \leq n-1 ((k_i, k_{i+1}) \in R \cup R^{-1}).$$

Ein R -Pfad zwischen zwei Knoten $k, k' \in K$ (in Zeichen: $R\text{-pfad}(k, k')$) ist ein Kantenzug $\{k_i\}_{i=1}^n (n > 0)$ mit:

$$\forall 1 \leq i, j \leq n (i \neq j \Rightarrow k_i \neq k_j) \wedge$$

$$\forall 1 \leq i \leq n-1 ((k_i, k_{i+1}) \in R) \wedge$$

$$(k_1 = k \wedge k_n = k').$$

R heit *zyklen- und schlingenfrei*, wenn es keinen R -Pfad $\{k_i\}_{i=1}^n (n \geq 2)$ gibt, mit: $k_1 = k_n$.

Eine Relation R heit *zusammenhngend*, wenn gilt:

$$\forall k, k' \in K (\exists k_1, k_2 \in K ((k, k_1), (k', k_2) \in R \cup R^{-1}))$$

$$\exists \text{ Kantenzug } \{e_i\}_{i=1}^n, n \geq 2 (k = e_1 \wedge k' = e_n).$$

II. Seien $M_i (1 \leq i \leq n)$ Mengen, $M = \prod_{i=1}^n M_i$ und $t = (m_1, \dots, m_n) \in M$.

Fr $1 \leq i \leq n$ sei $p_i(t)$ die i -te Projektion von t , d.h. $p_i(t) = m_i$.

III. Sei Z eine Menge.

Die Potenzmenge von Z wird notiert durch 2^Z .

IV. Sei $f:X \rightarrow Y$ Abbildung und $A \subseteq X$.

Die *Restriktion der Abbildung f auf A* wird notiert durch $f|_A$.

2.2 GRAPH-MODELL EINES MODULAREN PROGRAMMSYSTEMS (MODULGEFLECHT)

Grundlage der Beschreibung eines *Modulgeflechtes* sind *markierte Graphen* (Diagramme). Hinzu kommen *Zusatzbedingungen* für die *Kantenmenge*: Die *Knotenmarkierungen* werden Identifikationen von F- bzw. D-Moduln repräsentieren. Die *Kantenmarkierungen* dienen zur Darstellung der *Verflechtung* der Moduln bzw. der *Verteilung* der *Zugriffsrechte*.

Definition 2.1

Ein *Modulgraph* MG ist gegeben durch

$$MG = (K, R, M, \text{mark}, s),$$

wobei gilt:

- K ist die (endliche) Menge der *Knoten*(*bezeichnungen*),
- $R \subseteq K \times K$ ist eine zusammenhängende Relation zur Darstellung der *Kanten*,
- M ist das (endliche) *Markierungsalphabet* für die *Knoten* bzw. *Kanten*,
- $\text{mark} = (\text{kamark}, \text{knomark})$ ist ein Paar von *Markierungsfunktionen*,
- $s \in K$ ist ein ausgezeichneteter *Knoten*, der *Systemknoten*, mit folgenden Eigenschaften:

1. $K = K_F \cup K_D \cup \{s\}$, wobei

K_F bzw. K_D die Menge der F- bzw. D-Modul-Knoten ist.

2. $M = M_K \cup M_R$, wobei

$$M_K = M_F \cup M_D \cup \{\text{SYSTEM}\}^*),$$

*) Die Elemente von M_D sind im allgemeinen Paare, bestehend aus dem Namen eines D-Moduls und dessen "zugrunde liegender Art" (vgl. Kap. 3.1), z.B. (Y, keller).

$$M_R = \{dom, prov\} \cup (\{def, acc, prov\} \times 2^{\mathbb{Z}}).$$

M_F und M_D sind die Markierungsalphabete für die F- bzw. D-Modul-Knoten.

Z ist eine (endliche) Menge von Operationsbeschreibungen.

$$3. \text{ knomark} : K \rightarrow M_K,$$

$$\text{kamark} : R \rightarrow M_R, \text{ wobei gilt:}$$

$$\text{knomark}/_{K_D} \text{ ist injektiv}^*) \wedge$$

$$\forall f \in K_F (\text{knomark}(f) \in M_F) \wedge$$

$$\forall d \in K_D (\text{knomark}(d) \in M_D) \wedge$$

$$(\text{knomark}(s) = \text{SYSTEM}).$$

$$4. \exists_1 k \in K_U \{s\} ((s, k) \in R) \wedge \neg \exists k' \in K_U \{s\} ((k', k) \in R).$$

5. Sei

$$r_{FF} = \{(f, f') \in R \mid f, f' \in K_F\},$$

$$r_{FD} = \{(f, d) \in R \mid f \in K_F \wedge d \in K_D\},$$

$$r_{DD} = \{(d, d') \in R \mid d, d' \in K_D\},$$

$$r_{DF} = \{(d, f) \in R \mid d \in K_D \wedge f \in K_F\},$$

so gilt:

$$\forall kan \in r_{FF} \cup r_{DF} (\text{kamark}(kan) = dom) \wedge$$

$$\forall kan \in r_{FD} \cup r_{DD} (\text{kamark}(kan) \in \{def, acc\} \times 2^{\mathbb{Z}}) \wedge$$

$$(\text{kamark}(s, k) \in \{prov\} \cup (\{prov\} \times 2^{\mathbb{Z}})).$$

□

*) Für D-Moduln wird verlangt, daß sie eindeutig definiert sind, während F-Moduln mit dem gleichen Namen in mehreren Moduln als Untermodul spezifiziert sein können. Die Unterscheidung gleichmarkierter F-Moduln durch die Knotenbezeichnung dient zur Darstellung der spezifizierten F-Modul-Hierarchie.

In Figur 2/1 ist ein Beispiel für einen Modulgraphen angegeben.

Dort ist:

$$K_F = \{1, 2, 3\}, \quad K_D = \{4, 5\}$$

$$M_F = \{F_1, F_2, F_3\}, \quad M_D = \{D_1, D_2\}$$

$$R = \{(s, 1), (1, 2), (1, 4), (2, 4), (2, 5), (3, 5), (4, 3), (4, 5)\}$$

$$M_R = \{dom, prov\} \cup \{def, acc, prov\} \times 2^{\{OP_1, OP_2, OP_3, OP_4, OP_5\}}$$

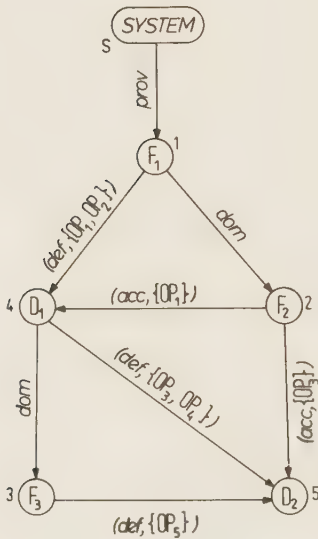


Fig. 2/1

Aus dem Beispiel in Figur 2/1 ist ersichtlich, daß durch die Definition 2.1 noch "unbrauchbare" Modulgraphen zugelassen sind:

Für den D-Modul D_2 gibt es zwei definierende Auftreten. Die Definitionen 2.2 bis 2.11 dienen unter anderem dazu, Situationen wie in Figur 2/1 zu verhindern.

Im folgenden ist für die verschiedenen Kantenarten in einem Modulgraphen eine Interpretation angegeben. Dabei sind (beispielhaft) spezielle Markierungen für Knoten und Kanten gewählt.

- Sei $(f, f') \in r_{FF}$ mit:

$knmark(f) = X \wedge$

$knmark(f') = Y \wedge$

$kamark((f, f')) = dom$

so bedeutet dies:

Der F-Modul X ist dem F-Modul Y übergeordnet, d.h. X dominiert Y .

Darstellung im Modulgraphen:



Analog ist eine Kante $(d, f) \in r_{DF}$ mit der Markierung dom zu interpretieren.

- Sei $(f, d) \in r_{FD}$ mit:

$knmark(f) = X \wedge$

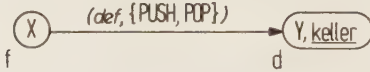
$knmark(d) = (Y, \underline{keller}) \wedge$

$kamark((f, d)) = (def, \{PUSH, POP\})$

so bedeutet dies:

Im F-Modul X ist ein D-Modul Y von der Art^{*)} keller mit den Zugriffsoperationen PUSH und POP *lokal* defi-niert.

Darstellung:



Analog ist eine Kante $(d, d') \in r_{DD}$ mit entsprechender Markierung zu interpretieren.

- Sei $(f, d) \in r_{FD}$ mit:

$knomark(f) = X \wedge$

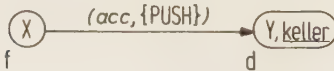
$knomark(d) = (Y, \underline{keller}) \wedge$

$kamark((f, d)) = (acc, \{PUSH\}),$

so bedeutet dies:

Dem F-Modul X wird Zugriff (engl.: access) auf einen D-Modul Y mit der Zugriffsoperation PUSH gewährt. Der D-Modul Y heißt *global* zum F-Modul X .

Darstellung:



- Sei $(d, d') \in r_{DD}$ mit:

$knomark(d) = (X, \underline{atype}) \wedge$

$knomark(d') = (Y, \underline{database}) \wedge$

$kamark((d, d')) = (acc, \{PUT, GET\}),$

so bedeutet dies:

Der D-Modul X hat Zugriff auf den D-Modul Y mit den Operationen PUT und GET.

^{*)} Zum Begriff "Art eines D-Moduls" vergleiche Kapitel 3.1.

Darstellung:



- Durch den Systemknoten s wird der "Haupt-Modul", d.h. der von einem Benutzer aufrufbare Modul eines Systems definiert^{*)}.

Sei $(s, d) \in R$ mit $d \in K_D$ und

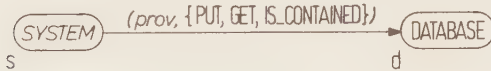
$$\text{knomark}(d) = \text{DATABASE} \wedge$$

$$\text{kamark}((s, d)) = (\text{prov}, \{\text{PUT}, \text{GET}, \text{IS_CONTAINED}\}),$$

so bedeutet dies:

Dem Benutzer wird eine Datenstruktur mit dem Namen DATABASE und den Zugriffsoperationen PUT, GET bzw. IS_CONTAINED zur Verfügung gestellt (engl.: provided).

Darstellung:



- Sei $(s, f) \in R$ mit $f \in K_F$ und

$$\text{knomark}(f) = \text{FTN_COMPILER} \wedge$$

$$\text{kamark}((s, f)) = \text{prov},$$

so bedeutet dies:

Dem Benutzer wird ein Programm mit dem Namen FTN_COMPILER zur Verfügung gestellt.

Darstellung:



*) Wegen Bedingung 4. in Def. 2.1 gilt stets, daß der mit Hilfe von s ausgezeichnete Knoten k bezüglich $R \setminus \{(s, k)\}$ ein maximaler Knoten ist.

Um die Komponenten der Markierungen von Kanten aus r_{FD} und r_{DD} bequemer selektieren zu können, seien für das weitere folgende mnemotechnische Abkürzungen definiert:

Sei $(k, k') \in r_{FD}^{Ur_{DD}}$, dann ist

- $\text{art}((k, k')) := p_1(\text{kamark}((k, k')))$
- $\text{zugriffe}((k, k')) := p_2(\text{kamark}((k, k')))$.

Die in Kapitel 1 vorgestellte Modularitäts-Philosophie verbietet eine beliebige Modulverflechtung bzw. Verteilung von Zugriffsrechten.

Im folgenden werden deshalb Einschränkungen bzgl. der zulässigen Kantenmenge bzw. der Kantenmarkierungen definiert.

Durch die Definitionen 2.2, 2.3 und 2.4 werden zunächst die Begriffe *F-Modul-* bzw. *D-Modul-Hierarchie* präzisiert. Die Definitionen 2.5 und 2.6 regeln, welche Verteilung von Zugriffsrechten in bezug auf diese Hierarchien zulässig ist. Nach einer formalen Definition des Prinzips "information hiding" dienen die Definitionen 2.8 bis 2.10 zur Charakterisierung des Begriffs "Systemhierarchie".

Definition 2.2

Ein Modulgraph heißt bezüglich der *F-Modul-Struktur hierarchisch*, wenn die durch r_{FF} definierte Relation eine Waldstruktur darstellt.

□

Definition 2.3

Ein Modulgraph heißt bezüglich der *F-Modul-Aufrufverschachtelung hierarchisch*, wenn die durch

$$\bar{r}_{FF} = \{ (fm_1, fm_2) \mid \exists (f_1, f_2) \in r_{FF} (fm_1 = \text{knemark}(f_1) \wedge fm_2 = \text{knemark}(f_2)) \}$$

definierte Relation zyklensfrei ist.

□

Durch mehrfaches Vorkommen von F-Moduln gleicher Markierung (d.h. gleichen Namens) entstehen beim Aufruf dieser F-Moduln Querverbindungen. Die Unterscheidung der Vorkommen durch die Knotenbezeichnungen ist durch die verschiedenen übertragenen Zugriffsrechte (vgl. Def. 2.13) motiviert.

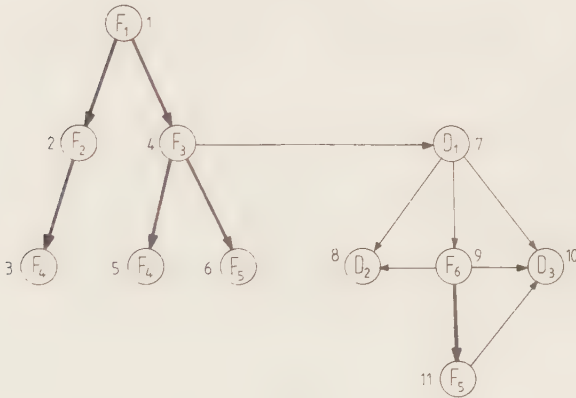


Fig. 2/2

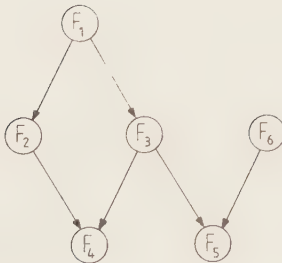


Fig. 2/3

In Figur 2/2 ist ein Beispiel für einen Modulgraphen mit hierarchischer F-Modul-Struktur und hierarchischer F-Modul-Aufrufverschachtelung angegeben. Dabei sind F-Modul-Knoten mit F_i , D-Modul-Knoten mit D_j markiert. Die Markierungen der Kanten sind der Übersicht wegen weggelassen. In Figur 2/3 ist die durch den Modulgraphen aus Figur 2/2 definierte Relation $\tilde{E}_{FF}$ dargestellt.

Die Definitionen 2.2 und 2.3 gewährleisten, daß durch die dynamische Aufrufverschachtelung *keine rekursiven* F-Modul-Aufrufe erfolgen. Beachtet man, daß das vorgestellte Konzept vornehmlich als Programmiermethode für "Programming-in-the-Large" (vgl. [DEK 76]) gedacht ist, so ist diese Forderung motiviert durch die Art der Parameter von F-Moduln - es sind nur Zugriffsberechtigungen auf D-Moduln zugelassen - und die unter Umständen sehr umfangreichen lokalen Daten von F-Moduln. Der Kellerungsaufwand wäre in diesem Fall selbst dann noch unververtretbar hoch, wenn man für die Parameter von F-Moduln generell "call by reference" vorsieht. Rekursive Aufrufverschachtelung ist somit nur vertretbar bei solchen F-Moduln, die ein Minimum an lokalen Datenstrukturen haben.

Eine rekursive Aufrufverschachtelung von *Prozeduren* ergibt sich z.B. im allgemeinen, wenn man für die Grammatik einer Programmiersprache ein Syntaxanalyseprogramm nach der Methode des rekursiven Abstiegs (vgl. [KNU 71]) entwirft. Betrachtet man die den nichtterminalen Symbolen der Grammatik zugeordneten Prozeduren als F-Moduln, so kann folgendes festgestellt werden:

- Die Schnittstelle der F-Moduln nach außen besteht aus einem lesenden Zugriff auf die Datei, welche das zu analysierende Quellprogramm enthält, und aus Zugriffen auf globale Listen (u.a. Liste der Begrenzer und

Symbolliste).

- Lokale Daten werden im wesentlichen keine benötigt.
- Die den F-Moduln zugrunde liegenden Algorithmen sind einfach und von geringem Umfang.

Die einzelnen so definierten F-Moduln (nicht das betrachtete Syntaxanalyseprogramm an sich) sind dem Bereich "Programming-in-the-Small" (vgl. [DEK 76]) zuzuordnen. Die in diesem Bereich (vernünftigerweise) zugelassene rekursive Aufrufverschachtelung steht jedoch nicht im Widerspruch zu dem für den Bereich "Programming-in-the-Large" gedachten Modul-Konzept, da dort rekursive Systemstrukturen ohnehin nicht der Regelfall sind.

Nachdem bereits Struktureigenschaften in bezug auf F-Moduln definiert wurden, sollen im weiteren Eigenschaften von Modulgraphen im Zusammenhang mit dem Aufbau von D-Modul-Strukturen (Datenstrukturen) angegeben werden.

Definition 2.4

Ein Modulgraph heißt bezüglich der *D-Modul-Struktur* *hierarchisch*, wenn gilt:

1. Die durch

$$r_{DD} = \{(d, d') \mid (d, d') \in r_{DD} \wedge \text{art}((d, d')) = \text{def}\}$$

definierte Relation besitzt Waldstruktur.

2. r_{DD} ist zyklensfrei.

□

Durch Definition 2.4 wird ein rekursiver Aufbau von Datenstrukturen verhindert. In Figur 2/4^{*)} ist eine D-Modul-Struktur angegeben, die dieser Definition genügt.

*) Ein "-" in der Markierung einer Kante bedeutet im weiteren jeweils, daß es auf die entsprechende Komponente nicht ankommt.

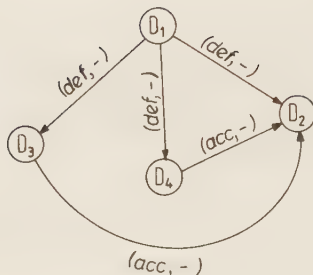


Fig. 2/4

Durch die Definitionen 2.2, 2.3 und 2.4 wird ein hierarchischer Aufbau von Modulstrukturen definiert. Im folgenden werden Eigenschaften von Modulgraphen im Zusammenhang mit der Weitergabe und Verteilung von Zugriffsberechtigungen auf D-Moduln angegeben. Die Eigenschaften beziehen sich auf Probleme der "Gültigkeit" bzw. "Sichtbarkeit" von Objekten in einem Modul (vgl. auch [KOS 76]).

Definition 2.5

Die Verteilung der Zugriffsrechte in einem Modulgraphen heißt *verträglich mit der F-Modul-Hierarchie*, wenn gilt:

$$\begin{aligned} & \forall (f, f') \in r_{FF} \quad \forall d \in K_D \\ & ((f', d) \in r_{FD} \wedge \text{art}((f', d)) = \text{acc}) \\ & \Rightarrow (f, d) \in r_{FD} \wedge \text{zugriffe}((f', d)) \subseteq \text{zugriffe}((f, d)). \end{aligned}$$

□

Die Bedingung gewährleistet, daß ein übergeordneter F-Modul (f) bzgl. eines D-Moduls (d) stets mehr Zugriffsrechte besitzt als ein *direkt* untergeordneter (f').

Zur Erläuterung von Definition 2.5 ist in Figur 2/5 ei-

ne typische Konstellation angegeben.

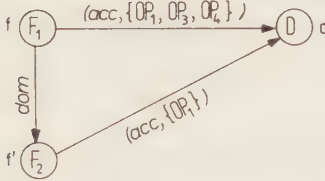


Fig. 2/5

Analog Definition 2.5 gelte bezüglich der D-Modul-Hierarchie $rdef_{DD}$:

Definition 2.6

Die Verteilung der Zugriffsrechte in einem Modulgraphen heißt *verträglich mit der D-Modul-Hierarchie*, wenn gilt:

$$\forall (d, d') \in rdef_{DD} \quad \forall d'' \in K_D$$

$$((d', d'') \in r_{DD} \wedge art((d, d'')) = acc$$

$$\Rightarrow (d, d'') \in r_{DD} \wedge$$

$$zugriffe((d', d'')) \subseteq zugriffe((d, d''))).$$

□

Die Einhaltung des Prinzips "information hiding" ist eine wesentliche Forderung für den Aufbau einer modularen Systemstruktur. Das Prinzip bedeutet grob gesprochen, daß die von einem Modul definierte interne Systemstruktur nach außen verborgen wird.

In unserem Graphmodell wird die Systemstruktur durch Kanten der Art *dom* bzw. *def* definiert.

Eine Präzisierung des Begriffes "information hiding" ist durch Definition 2.7 gegeben.

Definition 2.7

Sei $rsys$ die folgende Relation^{*)}:

$$rsys = \{ (k, k') \mid (k, k') \in R \wedge (kamark((k, k')) = dom \vee art((k, k')) = def) \}.$$

Sei weiter die Menge aller Nachfolgerknoten (engl. descendants) eines Knotens k unter der Relation $rsys$ definiert durch

$$desc(k) = \{ k' \mid (k, k') \in rsys^+ \}.$$

Ein Modulgraph heit *vertrglich mit dem Prinzip "information hiding"*, wenn gilt:

$$\begin{aligned} &\forall k, k', k'' \in K \\ &(k' \in desc(k) \wedge (k'', k') \in R \\ &\Rightarrow (k'' = k \vee k'' \in desc(k))). \end{aligned}$$

□

In Figur 2/6 ist eine durch Definition 2.9 ausgeschlossene Situation dargestellt (Kanten (k_1'', k_3') , $(k_2'' k_2')$). Die Menge $desc(k)$ ist gegeben durch $\{k_1', k_2', k_3'\}$.

*) $rsys$ beschreibt die Menge aller Kanten, die entweder mit dom oder mit $(def, -)$ markiert sind. Es gilt $rdef_{DD} \cup r_{FF} \subseteq rsys$.

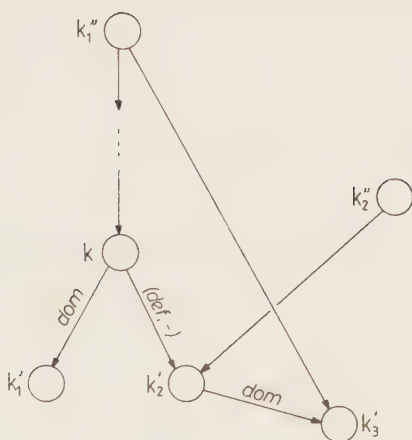


Fig. 2/6

Mit Hilfe der bisher definierten (lokalen) Bedingungen an Modulstrukturen werden im folgenden einige Charakterisierungssätze angegeben.

Die Eigenschaft, daß ein bzgl. r_{FF} oder $r_{def_{DD}}$ übergeordneter Modul stets mehr Zugriffsrechte zu einem D-Modul besitzt als ein *direkt* untergeordneter (vgl. Definition 2.5 bzw. 2.6), läßt sich übertragen auf r_{FF}^+ (Satz 2.1) bzw. $r_{def_{DD}}^+$ (Satz 2.2).

Satz 2.1

Für einen Modulgraphen M mit den Eigenschaften

- 1) die Verteilung der Zugriffsrechte in M ist verträglich mit der F -Modul-Hierarchie (Def. 2.5),

2) M ist verträglich mit dem Prinzip "information hiding" (Def. 2.7)

gilt:

$$\begin{aligned}
 & \forall (f, f') \in r_{FF}^+ \quad \forall d \in K_D ((f, d) \in r_{FD} \wedge (f', d) \in r_{FD}) \\
 \Rightarrow & (\forall f'' \in K_F ((f, f'') \in r_{FF}^+ \wedge (f'', f') \in r_{FF}^+)) \\
 \Rightarrow & (f'', d) \in r_{FD} \wedge \\
 & \quad \text{zugriffe}((f', d)) \subseteq \text{zugriffe}((f'', d)) \subseteq \text{zugriffe}((f, d)) \wedge \\
 & \quad \text{art}((f'', d)) = \text{art}((f', d)) = acc \wedge \\
 & \quad \text{art}((f, d)) \in \{def, acc\}.
 \end{aligned}$$

Die Aussage bedeutet folgendes:

Haben zwei F-Moduln (f und f'), die durch einen r_{FF} -Pfad miteinander verbunden sind, Zugriff zu einem D-Modul (d), so gilt dies auch für alle F-Moduln (f'') auf diesem Pfad. Darüber hinaus ist die Verteilung der Zugriffsrechte verträglich mit der F-Modul-Hierarchie. (*Verträglichkeit der Zugriffsverteilung auf einem r_{FF} -Pfad*).

Zum Beweis:

$$\begin{aligned}
 \text{i) Sei } (f, f') \in r_{FF}^+, \text{ so gilt:} \\
 & \forall f'' \in K_F ((f, f'') \in r_{FF}^+ \wedge (f'', f') \in r_{FF}^+) \\
 \Rightarrow & f'' \in R_{FF}\text{-pfad}(f, f') = \{f_i\}_{i=1}^n.
 \end{aligned}$$

ii) Nach Vor. 2) gilt:

$$\begin{aligned}
 & (f, d) \in r_{FD} \wedge (f', d) \in r_{FD} \wedge (f, f') \in r_{FF}^+ \\
 \Rightarrow & \text{kamark}((f', d)) \neq def \\
 \Rightarrow & \text{kamark}((f', d)) = acc.
 \end{aligned}$$

Mit i) und ii) folgt nun durch vollständige Induktion über die Länge von r_{FF} -pfad(f, f') unter Verwendung von Vor. 1) und 2):

$$\forall 1 \leq i \leq n-1 ((f_i, d) \in r_{FD} \wedge (f_{i+1}, d) \in r_{FD} \wedge$$

$$\text{zugriffe}((f_i, d)) \supseteq \text{zugriffe}((f_{i+1}, d)))$$

=> Behauptung. □

Figur 2/7 erläutert die Aussage von Satz 2.1. Hierbei gilt: $Z \supseteq Z'' \supseteq Z'$.

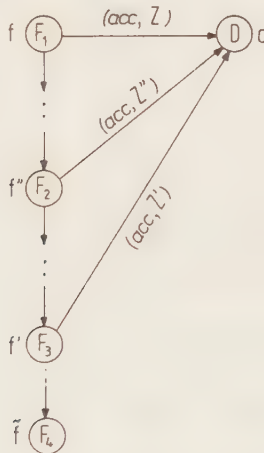


Fig. 2/7

Man beachte, daß im Gegensatz zur Blockstruktur der Gültigkeitsbereich eines Objektes (D-Moduls) nicht automatisch auf alle "tiefer liegenden" Programmeinheiten (Blöcke bzw. F-Moduln) ausgedehnt wird (vgl. Figur 2/7, F-Modul F_4). Darüber hinaus kann selbst bei Ausdehnung des Gültigkeitsbereichs eines D-Moduls die Zugriffsberechtig-

gung (d.h. die Menge der Zugriffsoperationen) eingeschränkt sein (vgl. Figur 2/5).

Satz 2.2

Die Verträglichkeit der Zugriffsverteilung auf einem $r\text{def}_{\text{DD}}$ -Pfad gilt analog zu Satz 2.1, unter der Voraussetzung von Definition 2.6 und 2.7. □

Ein weiterer wesentlicher Aspekt des in Kapitel 1 vorgestellten Modul-Konzeptes ist die Forderung nach einer *zentralen Ressourcen-Verwaltung*. Sie bedeutet das Verbot der Kommunikation untergeordneter Moduln, ohne Kontrolle durch den übergeordneten.

Hierzu gilt der folgende Satz:

Satz 2.3 (Ressourcen-Verträglichkeit)

In einem Modulgraphen, der dem Prinzip "information hiding" genügt, ist die *Verteilung der Zugriffsberechtigungen verträglich mit dem Prinzip der zentralen Ressourcen-Verwaltung*, d.h. es gilt:

Sei $r = r\text{def}_{\text{DD}} \vee r = r_{\text{FF}}$, dann folgt:

$$\forall k, k', k'' \in K \forall d \in K_D$$

$$((k, k') \in r \wedge (k, k'') \in r \wedge k' \neq k'' \wedge (k', d) \in r \wedge \text{art}((k', d)) = \text{def}) \\ \Rightarrow (k'', d) \notin r.$$

Die Aussage folgt unmittelbar aus Definition 2.7. □

In Figur 2/8 ist eine durch die Aussage von Satz. 2.3 ausgeschlossene Situation dargestellt.

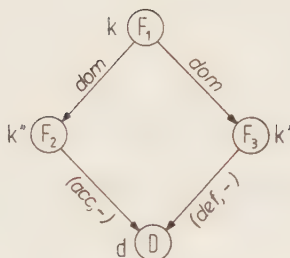


Fig. 2/8

In Kapitel 1 wird in den Definitionen 1.5 bzw. 1.6 gefordert, daß die Schnittstelle eines Moduls nach innen die Spezifikation *untergeordneter F-* bzw. die Definition *lokaler D-Moduln* enthält.

Die beiden folgenden Definitionen gewährleisten, daß alle Moduln, bis auf den durch den Systemknoten s ausgezeichneten, in genau einem Modul spezifiziert bzw. definiert sind.

Definition 2.8

Die F -Moduln eines Modulgraphen heißen *wohlspezifiziert*, wenn gilt:

$$\forall f \in K_F ((s, f) \in R \Rightarrow$$

$$\exists_1 k \in K ((k, f) \in r_{FF} \cup r_{DF}))^*).$$

□

*) Wegen Definition 2.1 gilt: $\text{kamark}((k, f)) = \text{dom}$.

Definition 2.9

Die D-Moduln eines Modulgraphen heißen *wohldefiniert*, wenn gilt:

$$\forall d \in K_D ((s, d) \in R \Rightarrow \exists_1 k \in K((k, d) \in r_{FD} \cup r_{DD} \wedge \text{art}((k, d)) = \text{def})).$$

□

Wir betrachten nun noch einmal die in Definition 2.7 angegebene Relation r_{sys} , die im weiteren als *Systemhierarchie* bezeichnet werde.

Für r_{sys} gilt die folgende Aussage:

Satz 2.4

Unter den Voraussetzungen, daß r_{FF} und r_{DD} Waldstruktur besitzen, hat r_{sys} genau für wohlspezifizierte, wohldefinierte Modulgraphen Baumstruktur.

Der Beweis folgt unmittelbar aus der Definition von r_{sys} (vgl. Definition 2.7). Die Wurzel von r_{sys} ist der durch den Systemknoten s ausgezeichnete Knoten.

□

In Kapitel 1 wurde zwischen funktions- und datenorientierten Systemen unterschieden.

Die folgende Definition präzisiert diese Begriffe.

Definition 2.10

Sei r_{sys} baumstrukturiert, $(s, k) \in R$ und k die Wurzel des Baumes.

Ein Modulgraph heißt

- *funktionsorientiert*, wenn $k \in K_F$,
- *datenorientiert*, wenn $k \in K_D$ gilt.

□

Figur 2/9 zeigt einen funktionsorientierten, Figur 2/10 einen datenorientierten Modulgraphen.

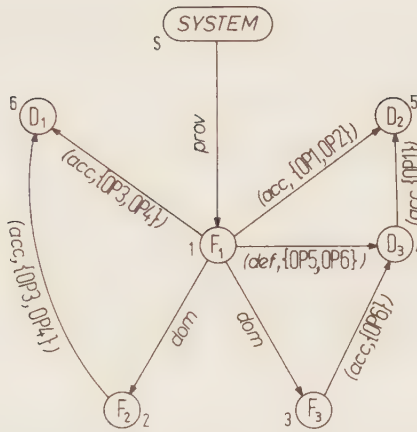


Fig. 2/9

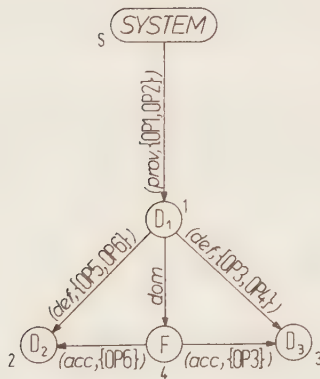


Fig. 2/10

Die Systemhierarchie r_{sys} setzt sich wie folgt zusammen:

$$r_{sys} = r_{FF} \cup r_{DD} \cup r_{DF} \cup r_{FD},$$

wobei r_{DF} gegeben ist durch

$$r_{FD} = \{kan \mid kan \in r_{FD} \wedge art(kan) = def\}.$$

In den Definitionen 2.5 bzw. 2.6 ist bereits gefordert, daß die Verteilung der Zugriffsrechte verträglich ist mit den Relationen r_{FF} bzw. r_{DD} . Verlangt man dies zusätzlich auch in bezug auf die Relationen r_{DF} und r_{FD} , so ergibt sich die Verträglichkeit bezüglich r_{sys} .

Definition 2.11

Die Verteilung der Zugriffsrechte in einem Modulgraphen M heißt *verträglich mit der Systemhierarchie r_{sys}* , wenn gilt:

- Die Verteilung ist verträglich mit der F- und D-Modul-Hierarchie
- $\forall (k, k') \in r_{FD} \cup r_{DF} \forall d \in K_D ((k', d) \in r_{FD} \wedge art((k', d)) = acc \Rightarrow (k, d) \in r_{FD} \wedge zugriffe((k', d)) \subseteq zugriffe((k, d)))$

□

Die einen Modulgraphen charakterisierenden Definitionen 2.2 bis 2.11 können nun dazu verwendet werden, die Begriffe "konsistenter Modulgraph" bzw. "Modulgeflecht" einzuführen.

Definition 2.12

Ein Modulgraph M heißt *konsistent* oder *Modulgeflecht*, wenn gilt:

- M ist bezüglich der F-Modul-Struktur, der F-Modul-Aufrufverschachtelung und der D-Modul-Struktur hierarchisch (Def. 2.2, 2.3, 2.4).
- M ist verträglich mit dem Prinzip "information hiding" (Def. 2.7).

- M ist funktions- oder datenorientiert (Def. 2.10).
- Die Verteilung der Zugriffsrechte in M ist vertraglich mit der Systemhierarchie (Def. 2.11).

□

Ausgehend von der formalen Beschreibung eines modularen Programmsystems durch ein Modulgeflecht, kann man formale Beschreibungen für F- und D-Moduln ableiten, indem man die über eine Kante direkt erreichbare Umgebung der F- bzw. D-Modul-Knoten betrachtet.

Die Beschreibung richtet sich dabei nach den in Kapitel 1.3 gemachten Vorschlägen zum Aufbau der Moduln.

Zur Definition der Moduln wird noch der Begriff der *an einen Modul von außen übertragenen Zugriffsrechte* (vgl. Def. 1.5 u. 1.6, S. 11 bzw. 12).

Definition 2.13

Sei k ein Modulknoden. Die *an k von außen übertragenen Zugriffsrechte* (in Zeichen $\dot{U}ZR(k)$) sind definiert durch:

$$\dot{U}ZR(k) = \{ \langle d, z_d \rangle \mid (k, d) \in r_{FD}^{Ur_{DD}} \wedge art((k, d)) = acc \\ \wedge z_d = zugriffe((k, d)) \}.$$

□

Stellt man sich im Zusammenhang mit einer programmiersprachlichen Notation von Modulgeflechten vor, daß die an einen Modul k übertragenen Zugriffsrechte als *Parameter* übergeben werden, so charakterisiert $\dot{U}ZR(k)$ die Menge *aller dynamisch möglichen, aktuellen* (Parameter-) Bindungen von k zu D-Moduln. Die beiden folgenden Definitionen für F- bzw. D-Moduln tragen der Forderung nach Kontextunabhängigkeit (vgl. Def. 1.1, S. 7) nicht Rechnung. Es wird vielmehr die "Verankerung" eines Moduls im Gesamtsystem beschrieben. Zur Formulierung von formalen (kontextunabhängigen) Schnittstellen nach außen wäre eine Erweiterung des Graphmodells um "formale D-Modul-Kno-

ten" nötig. Da umgebungsunabhängige Darstellungen von Modulen durch die in Kapitel 3 eingeführte programmiersprachliche Notation möglich sind, wird auf die obenerwähnte Erweiterung verzichtet.

Definition 2.14

Sei $d \in K_D$. Der durch d definierte D -Modul dm wird vollständig beschrieben durch:

$$dm = (id, Zop, \ddot{U}ZR, D, F)$$

wobei

$$id = knomark(d)$$

$$Zop = \text{zugriffe}((k, d)), \text{ mit } (k, d) \in R \wedge \text{art}((k, d)) \in \{def, prov\}$$

$$\ddot{U}ZR = \ddot{U}ZR(d)$$

$$D = \{((d', z'), \ddot{U}ZR(d')) \mid (d, d') \in r_{DD}^{def} \wedge z' = \text{zugriffe}((d, d'))\}$$

$$F = \{(f, \ddot{U}ZR(f)) \mid (d, f) \in r_{DF}\}.$$

id ist die Identifikation des D -Moduls.

Zop ist die Menge der auf dm definierten Zugriffsoperationen.

$\ddot{U}ZR$ charakterisiert die von außen an dm übertragenen Zugriffsrechte.

D charakterisiert die "innerhalb" von dm lokal definierten (benutzten) D -Moduln.

F charakterisiert die lokal definierten (benutzten) F -Moduln.

□

Betrachtet man in Figur 2/11 den durch den Knoten 4 definierten D -Modul, so ergibt sich:

$$id = D_1$$

$$Zop = \{OP1, OP2\}$$

$$\ddot{U}ZR = \{(3, \{OPO\})\}$$

$$D = \{((6, \{OP6, OP7\}), \emptyset), ((8, \{OP4, OP5\}), \emptyset)\}$$

$$F = \{(7, \{(6, \{OP7\}), (7, \{OP5\})\})\}.$$

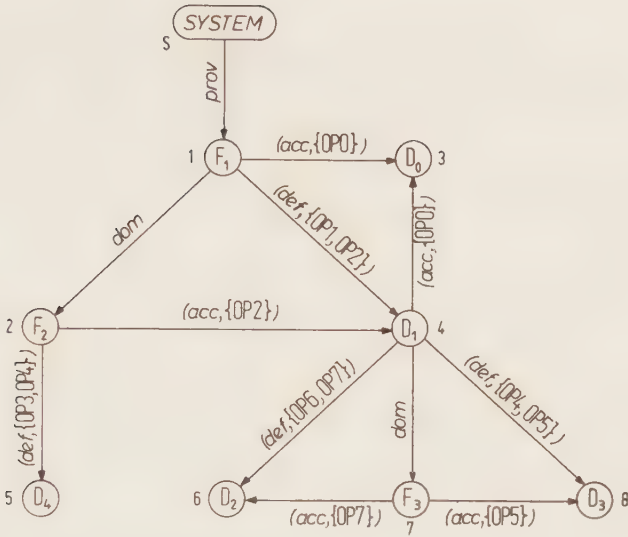


Fig. 2/11

Definition 2.15

Sei $f \in K_F$. Der durch f definierte F -Modul fm wird vollständig beschrieben durch:

$$fm = (id, \dot{U}ZR, D, F),$$

wobei

$$id = knomark(f)$$

$$\dot{U}ZR = \dot{U}ZR(f)$$

$$D = \{ ((d, Z'), \dot{U}ZR(d)) \mid (f, d) \in r_{FD} \wedge Z' = \text{zugriffe}((f, d)) \}$$

$$F = \{ (f', \dot{U}ZR(f')) \mid (f, f') \in r_{FF} \}.$$

id ist der Identifikator von fm.

ÜZR charakterisiert die von außen an fm übertragenen Zugriffsrechte.

D und F sind analog wie bei D-Moduln.

□

Betrachtet man in Figur 2/9 den durch den Knoten 1 definierten F-Modul fm, so ergibt sich:

id = F1

ÜZR = $\{(6, \{OP3, OP4\}), (5, \{OP1, OP2\})\}$

D = $\{((4, \{OP5, OP6\}), ((5, \{OP1\})))\}$

F = $\{(2, \{(6, \{OP3, OP4\})\}), (3, \{(4, \{OP6\})\})\}$.

Die vorangegangenen Definitionen dienen zur formalen Beschreibung eines modularen Programmsystems sowohl in seiner Gesamtheit als Modulgeflecht als auch der Einzelkomponenten.

Das vorgestellte Modell beinhaltet die Darstellung aller dynamisch möglichen Vernetzungen der Moduln untereinander. Es geht nicht hervor, welche aktuellen Verbindungen zur Laufzeit bestehen können.

Die in Kapitel 3 zu entwerfende programmiersprachliche Notation eines modularen Systems sieht die Übertragung von Zugriffsrechten auf Parameterpositionen der Moduln vor. Aus der Markierung der Zugriffskanten ist jedoch nicht ersichtlich, welcher Zugriff zu welcher Parameterposition gehört. Dies hätte man beispielsweise durch eine Erweiterung der Markierung der Zugriffskanten um eine zusätzliche Komponente (Parameterposition) erreichen können.

Eine andere Möglichkeit besteht darin, für die übertragenen Zugriffsrechte (vgl. Def. 2.13) eine Parametrisierung zu definieren.

Definition 2.16

Sei $k \in K, \dot{U}ZR(k)$ die an k übertragenen Zugriffsrechte.

Eine *Parametrisierung* p von $\dot{U}ZR(k)$ ist ein Paar

$$(\text{param}_k, n_k),$$

wobei

$n_k \in \mathbb{N}$ (die Anzahl der "Zugriffsparameter" von k)^{*)}

$$\text{param}_k : \dot{U}ZR(k) \xrightarrow{\text{surj}} \{1, \dots, n_k\}.$$

Für param_k gilt:

$$(+)\ \forall (d, z_d), (d', z_{d'}) \in \dot{U}ZR(k)$$

$$(z_d \neq z_{d'} \Rightarrow \text{param}_k((d, z_d)) \neq \text{param}_k((d', z_{d'}))).$$

□

Durch eine Parametrisierung wird somit jeder möglichen Bindung eines Moduls zu D-Moduln bzw. jeder der zugrunde liegenden Zugriffskanten eine Parameterposition zugeordnet. Die Zuordnung erfolgt derart, daß verschiedene Bindungen bzw. verschieden markierte Zugriffskanten auf verschiedene Parameterpositionen abgebildet werden (Bed. (+)).

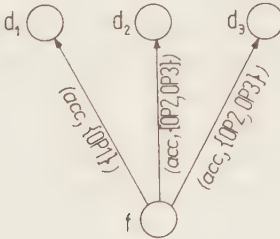


Fig 2/12

*) Bei D-Moduln sind die Zugriffsparameter zu unterscheiden von den Kreierungsparametern bzw. den Parametern der Operationen (vgl. Def. 1.5, Kap. 1.3).

Beispiel 2.1

In Figur 2/12 sind die an einen F-Modul f übertragenen Zugriffsrechte dargestellt. Hierfür ergeben sich unter anderem folgende Möglichkeiten der Parametrisierung:

1. $n_f=2$

$$\text{param}_f = \{ ((d_1, \{OP_1\}), 1), \\ ((d_2, \{OP_2, OP_3\}), 2), \\ ((d_3, \{OP_2, OP_3\}), 2) \}$$

2. $n_f=3$

$$\text{param}_f = \{ ((d_2, \{OP_2, OP_3\}), 1), \\ ((d_1, \{OP_1\}), 2) \\ ((d_3, \{OP_2, OP_3\}), 3) \}$$

□

Das vorgestellte Modul-Konzept sieht die Verwendung *vordefinierter* F- und D-Moduln vor. Im herkömmlichen programmiersprachlichen Sinn ist dies am besten zu vergleichen mit der Verwendung von Standard-Prozeduren und -Arten (z.B. der Modus integer in ALGOL 68).

Für den Modulgraphen bedeutet dies, daß es verschiedene F-Modul-Knoten mit gleicher Markierung geben kann. Bei D-Modul-Knoten kann dies nicht auftreten. Es kann zwar verschiedene D-Moduln geben, denen die gleiche (abstrakte) Art zugrunde liegt, die sich aber dennoch durch verschiedene Objektnamen unterscheiden, so daß die Markierung verschiedener D-Modul-Knoten jeweils verschieden ist.

Beim Aufbau eines konsistenten Modulsystems ist nun darauf zu achten, daß für vordefinierte Moduln die übertragenen Zugriffsrechte "ähnlich" und die Parametrisierungen "miteinander verträglich" sind. Das vorgestellte Modell läßt eine formale Beschreibung dieses Sachverhaltes zu. Hierauf wird jedoch verzichtet, da die in Kapitel 3 ent-

worfene programmiersprachliche Notation ohnehin eine, verglichen mit dem Graphmodell, verfeinerte Darstellung eines modularen Systems ermöglicht.

Im folgenden werden noch einige Begriffe eingeführt, die zum Teil bereits in ihrer intuitiven Bedeutung verwendet wurden.

- Sei $dm = (id, Zop, \dot{U}ZR, D, F)$ ein D-Modul.

1. Das Paar (id, Zop) heißt die von dm zur Verfügung gestellte Schnittstelle nach außen.
2. $\dot{U}ZR$ definiert die von außen übertragene Schnittstelle.
3. D und F definieren zusammen die Schnittstelle von dm nach innen.
4. Ein Paar (id, op) mit $op \in Zop$ heißt Zugriffsrecht bezüglich des D-Moduls dm .

- Sei $fm = (id, \dot{U}ZR, D, F)$ ein F-Modul.

1. Das Paar $(id, \dot{U}ZR)$ heißt Schnittstelle von fm nach außen.
2. Durch $\dot{U}ZR$ und D werden die dem F-Modul fm zur Verfügung stehenden Ressourcen definiert.

2.3 DARSTELLUNG VON MODULGEFLECHTEN

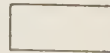
Die bisherige Darstellung von Modulgeflechten kann bei umfangreicheren Gebilden durch die Vielzahl von Markierungen und der Kreuzung von Kanten sehr unübersichtlich werden.

Im folgenden wird (durch die Vereinbarungen 1 - 4) ein Vorschlag zur übersichtlichen Darstellung von Modulgeflechten in schaltplanähnlicher Form gemacht. Die Grundidee dabei ist, daß jeweils nur ein lokaler Aspekt eines Moduls gezeichnet wird. Im wesentlichen ist dies die in den Definitionen 2.14 bzw. 2.15 zur Beschreibung von Moduln angegebene Information. Zugriffskanten werden durch die Einführung von Hilfsknoten "aufgeschnitten". Die so entstehenden Knoten spielen dann die Rolle von "Anschlußpunkten".

Um dennoch globalere Zusammenhänge darstellen zu können, wird die Verwendung von strukturellen Modulgeflechten (siehe Vereinbarung 5) empfohlen.

Vereinbarung 1 (Form der Knoten)

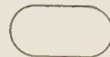
- F-Modul-Knoten



- Knoten vordefinierter
F-Moduln



- D-Modul-Knoten



- Knoten vordefinierter
D-Moduln



- Der durch den Systemknoten s ausgezeichnete Knoten wird durch eine Doppelumrandung $\equiv$ gekennzeichnet

net. Der Knoten s selbst wird *nicht* gezeichnet.

- Die Markierungen werden innerhalb der Umrandung notiert.
- Knotenbezeichnungen werden weggelassen, da eine Unterscheidung der Knoten durch das Auftreten selbst gegeben ist.

□

Vereinbarung 2 (*Art der Kanten*)

- Kanten k mit $\text{art}(k)=\text{dom}$
(Untermodulkanten)
- Kanten k mit $\text{art}(k)=\text{def}$
(Definitions-kanten)
- Kanten k mit $\text{art}(k)=\text{acc}$
(Zugriffskanten)
- Die Kante k mit $\text{art}(k)=\text{prov}$ wird nicht explizit gezeichnet, da sie im Graphmodell nur zur Auszeichnung des "Hauptmoduls" eines Systems dient, was in dieser Darstellung durch eine spezielle Knotenform geschieht.

□

Im weiteren werden nun noch Vereinbarungen getroffen, die im wesentlichen das "Aufschneiden" von Zugriffskanten im Zusammenhang mit der Darstellung von F- und D-Moduln gemäß den Definitionen 2.14 bzw. 2.15 betreffen.

Vereinbarung 3 (*Darstellung eines D-Moduls*)

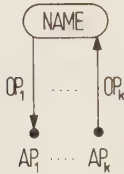
Sei $\text{dm} = (\text{id}, \text{Zop}, \text{ÜZR}, \text{D}, \text{F})$ ein durch einen Knoten d definierter D-Modul.

Die Darstellung zerfällt in vier Komponenten:

- Zur Verfügung gestellte Schnittstelle nach außen
(id, Zop)
- Übertragene Zugriffsrechte (ÜZR)
- Definitionskanten (D)
- Untermodulkanten (F).

1. Zur Verfügung gestellt Schnittstelle nach außen

Sei $(NAME, \{OP_1, \dots, OP_k\})$ die zur Verfügung gestellte Schnittstelle nach außen. Die Darstellung ist wie folgt*):

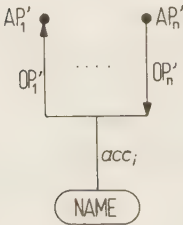


Bemerkung:

Die Hilfsknoten (*Anschlußpunkte*) AP_i ($1 \leq i \leq k$) dienen dazu, jede Zugriffskante mit der Markierung (acc, Z) mit $Z \subseteq \{OP_1, \dots, OP_k\}$ darstellen zu können (vgl. Beispiel 2.2).

2. Übertragene Zugriffsrechte

Sei $(d', \{OP'_1, \dots, OP'_n\}) \in \mathcal{UZR}$, so wird die zugrunde liegende Zugriffskante wie folgt dargestellt:

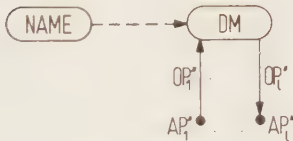


*) Die Pfeilrichtungen sollen dabei andeuten, ob es sich um einen lesenden ($\xrightarrow{OP_1} AP_1$) oder schreibenden ($\xleftarrow{OP_k} AP_k$) Zugriff handelt.

Hierbei sind die AP_i' ($1 \leq i \leq n$) die durch d' , entsprechend Vereinbarung 3/1, definierten Anschlußpunkte. Legt man eine Parametrisierung zugrunde, so bedeutet acc_i' , daß es sich um die i -te Parameterposition handelt.

3. Definitionskanten

Sei $(d', \{OP_1'', \dots, OP_l''\}) \in D$ und $knmark(d') = DM$. Die zugrunde liegende Zugriffskante wird wie folgt dargestellt:



4. Untermodulkanten

Sei $(f, \text{ÜZR}(f)) \in F$ und $knmark(f) = FM$. Die zugrunde liegende Untermodulkante wird wie folgt dargestellt:



□

Vereinbarung 4 (Darstellung eines F-Moduls)

Die Darstellung eines F-Moduls

$$fm = (id, \text{ÜZR}, D, F)$$

zerfällt in drei Komponenten:

- Schnittstelle nach außen ($id, \text{ÜZR}$)
- Definitionskanten (D)
- Untermodulkanten (F).

Die Darstellung erfolgt analog Vereinbarungen 3/2., 3/3. und 3/4..

Beispiel 2.2

Im folgenden wird die Umsetzung der durch Figur 2/13 gegebenen Modulstruktur in eine Darstellung gemäß den Vereinbarungen 1 - 4 (Figur 2/14) erläutert.

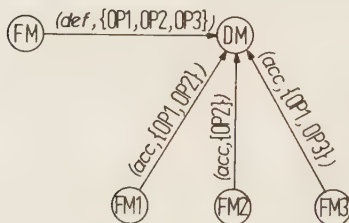


Fig. 2/13

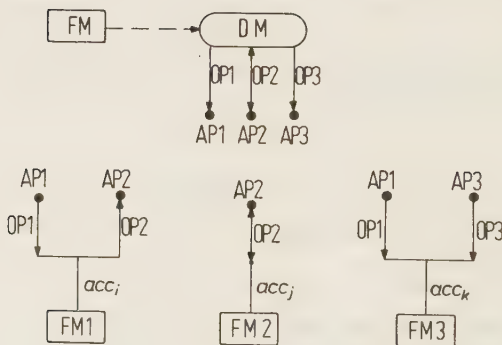


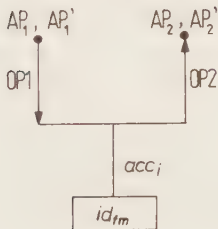
Fig. 2/14

Wird für die übertragenen Zugriffsrechte eine Parametrisierung zugrunde gelegt, so werden *verschiedene* Zugriffskanten zu *einer* Parameterposition durch entsprechend verschiedene Anschlußpunkte wiedergegeben.

Beispiel 2.3 erläutert diese Situation.

Beispiel 2.3

Seien $(d, \{OP1, OP2\})$ und $(d', \{OP1, OP2\})$ aus $\mathcal{UZR}_{fm}$. Werden beide Zugriffskanten zur gleichen Parameterposition i gezählt, so kann dies folgendermaßen dargestellt werden:



AP_1 bzw. AP_1' und AP_2 bzw. AP_2' sind dabei jeweils einander entsprechende Anschlußpunkte der durch d bzw. d' definierten D-Moduln.

□

Bei der Darstellung eines Modulgeflechtes ist im allgemeinen wie folgt vorzugehen:

- Man betrachtet alle F- bzw. D-Modul-Knoten, von denen ' aus Definitions- bzw. Untermodulkanten ausgehen (d.h. nicht Endknoten in dem Modulgeflecht sind).
- Die F- bzw. D-Moduln, die durch diese Knoten (gemäß den Definitionen 2.10 und 2.11) definiert sind, werden in

einer der Hierarchie der Knoten entsprechenden Reihenfolge dargestellt.

- Bei den D-Moduln ist insgesamt darauf zu achten, daß die Namen der Anschlußpunkte eindeutig gewählt werden.

Die Darstellung eines Modulgeflechtes gemäß den Vereinbarungen 1 - 4 spiegelt zwar die globale Verflechtung eines einzelnen Moduls im Gesamtsystem wider, gibt aber keinen Einblick in globale Zusammenhänge. Als Ergänzung wird deshalb durch Vereinbarung 5 eine Gesamt- oder teilweise Darstellung eines Modulgeflechtes vorgeschlagen.

Vereinbarung 5 (*strukturelles Modulgeflecht*)

Ein *strukturelles Modulgeflecht* ist die Gesamt- oder Teildarstellung des zugrunde liegenden Modulgraphen in folgender Form:

- Alle Kantenmarkierungen werden weggelassen. Es wird nur die Art der Kanten gemäß Vereinbarung 2 gezeichnet.
- Die Form der Knoten und deren Markierung ergibt sich aus Vereinbarung 1.
- Es werden keine Kanten "aufgeschnitten".

□

Ein Beispiel hierzu findet man im nachfolgenden Abschnitt (Fig. 2/15, S. 64).

2.4 ANWENDUNGSBEISPIELE

Zur Erläuterung des in Kapitel 1 vorgestellten Modul-Konzeptes und zur Demonstration der formalen Beschreibung bzw. der entsprechenden bildlichen Darstellung werden im folgenden zwei Beispiele angegeben.

Das erste befaßt sich mit einer Problemstellung aus dem Bereich der Prozeßrechner-Software. Das zweite Beispiel befaßt sich mit der Strukturierung eines datenorientierten Modulsystems.

Es kommt hierbei weniger auf das Verständnis der Funktion der einzelnen Moduln als auf die Darstellung der möglichen Interaktion der Moduln an.

Beispiel 2.4

Das folgende Beispiel betrachtet eine Problemstellung aus dem Bereich der Prozeßrechner-Software. Die diesem Beispiel zugrunde liegende Terminologie ist im wesentlichen aus [REI 77] und [SIM 76] entnommen.

Obwohl der Ausgangspunkt der Überlegungen im Bereich der sequentiellen Programmierung liegt, soll gerade an diesem Beispiel gezeigt werden, daß sich die Methode durchaus eignet, die Struktur (nicht etwa das dynamische Verhalten) eines Software-Systems aus dem Bereich der parallelen Programmierung zu beschreiben.

Problemstellung

Es ist ein System zur Steuerung einer Produktionsanlage zu entwerfen. Es sei bekannt, daß die Anlage in drei Verarbeitungskomponenten zerfällt: *Vorverarbeitung*, *Feinverarbeitung* und *Nachbereitung*. Zu verwalten ist eine Reihe von komponentenspezifischen sowie zentralen Analog- und Binäreingängen. Das analoge Verhalten der Anlage soll nach gewissen Gesichtspunkten zur späteren

Untersuchung archiviert werden.

Bei der Zerlegung des Systems in Moduln ist zu berücksichtigen, daß eine Reihe vordefinierter Moduln (Grundsoftware) gegeben ist:

- *analoge Meßwerterfassung*
(abgekürzt: ANERF)
- *Digitaleingabe*
(abgekürzt: DIGEIN)
- *Meßwertarchivierung*
(abgekürzt: MESSARCH)
- *spezielle Steuerungsmechanismen*
(abgekürzt: STEUER).

Entsprechend der Vorgehensweise bei der modularen Programmentwicklung (vgl. Kapitel 1.4) werden im folgenden Entwurfsentscheidungen (die einer gewissen Willkür unterliegen) getroffen und diesen Entscheidungen jeweils ein Modul zugeordnet:

- Die Steuerung der Gesamtanlage wird von einem F-Modul "*Produktionssteuerung*" (abgekürzt durch PROD-STEUER) übernommen.
- Der Zustand aller binären Eingänge wird in einem (globalen) D-Modul "*Prozeßabbild*" (abgekürzt PROZABB) gehalten. Das Prozeßabbild gibt zu jedem Zeitpunkt Auskunft über den Gesamtzustand der Anlage.
- Bei den analogen Eingängen erweist es sich als günstig, zu unterscheiden zwischen zentral zu bearbeitenden und denen, die den drei Verarbeitungskomponenten zuzuordnen sind. Das Verhalten der *zentralen Analogeingänge* sowie Analoginformation über den Zustand der Teilkomponenten werden in dem D-Modul "*zentrale Analogdaten*" (abgekürzt ZENANDAT) gehalten.

ten. Die komponentenspezifischen Analogeingänge werden in den D-Moduln

- "*Analogdaten Vorverarbeitung*"
(abgekürzt ANDATVOR)
- "*Analogdaten Feinverarbeitung*"
(abgekürzt ANDATFEIN)
- "*Analogdaten Nachbereitung*"
(abgekürzt ANDATNACH)

bearbeitet.

- Durch die Gliederung der Anlage in drei Teilkomponenten bietet sich eine Realisierung derselben durch die F-Moduln

- "*Vorverarbeitung*" (abgekürzt VORVERARB)
- "*Feinverarbeitung*" (abgekürzt FEINVERARB)
- "*Nachbereitung*" (abgekürzt NACHBEREIT)

an.

Zur Darstellung der gegenseitigen Verflechtung wird im folgenden eine kurze Beschreibung der Funktionen der einzelnen Moduln angegeben.

- Beschreibung des F-Moduls "Produktionssteuerung"

Die Koordinierung der Bearbeitung der D-Moduln *Prozeßabbild* und *zentrale Analogdaten* erfolgt im zentralen Steuermodul *Produktionssteuerung*.

Das System wird nun so konzipiert, daß die zyklischen und/oder azyklischen Aufrufe der F-Moduln *analoge Meßwerterfassung*, *Digitaleingabe*, *Vorverarbeitung*, *Feinverarbeitung* und *Nachbereitung* in den D-Moduln *Prozeßabbild* und *zentrale Analogdaten* Werte zurückliefern (d.h. es müssen entsprechende schreibende

Zugriffe ermöglicht werden), die von dem F-Modul *spezielle Steuerungsmechanismen* interpretiert werden und zur entsprechenden Digitalausgabe (über *Prozeßabbild*) führen.

Es besteht keine Notwendigkeit der Kommunikation zwischen dem Modul *Produktionssteuerung* und den untergeordneten F-Modulen *Vorverarbeitung*, *Feinverarbeitung* und *Nachbereitung* in bezug auf Binäreingänge, da diese zentral verwaltet werden.

Die Daten im D-Modul *zentrale Analogdaten* werden zyklisch durch Aufrufe des F-Moduls *Meßwertarchivierung* dokumentiert.

- Beschreibung des F-Moduls *Digitaleingabe*

Der vordefinierte Modul bearbeitet geräteabhängige Information. Er hat die Aufgabe, die Eingaben der digitalen Meßstellen in den D-Modul *Prozeßabbild* zu übertragen. Erreicht wird dies beispielsweise dadurch, daß dem Modul *Digitaleingabe* die Adressen der digitalen Meßstellen (Binäreingänge) bekannt gemacht werden.

- Beschreibung des F-Moduls *analoge Meßwerterfassung*

Der Modul *analoge Meßwerterfassung* arbeitet ähnlich wie der Modul *Digitaleingabe*. Durch Übergabe von Adressen analoger Meßstellen kann eine zyklische oder azyklische Meßwerterfassung angestoßen werden.

Bei der Realisierung erweist es sich als günstig, einen geräteabhängigen Modul *zentrales Leseprogramm* (abgekürzt ZENLES) zu verwenden. Zur Kommunikation mit diesem dient ein (vordefinierter) D-Modul *Rohwertliste* (abgekürzt ROHLIS). Die vom F-Modul *analoge Meßwerterfassung* in den D-Modul *Rohwertliste* eingetragenen Lesewünsche werden von dem Modul *zentra-*

les Leseprogramm befriedigt.

- Beschreibung der F-Moduln *Vorverarbeitung, Feinverarbeitung und Nachbereitung*

Diese Moduln bearbeiten die komponentenspezifischen Analogdaten in den entsprechenden D-Moduln. Das Abspalten der komponentenspezifischen von den zentralen Analogdaten (D-Modul ZENANDAT) wurde vorgenommen, um in den Teilkomponenten andere Bearbeitungszyklen zu ermöglichen als im zentralen Steuerprogramm. Die Teilkomponenten liefern im D-Modul *zentrale Analogdaten* "Sammelinformation" zur weiteren Verarbeitung an den Modul *Produktionssteuerung* ab.

Auf die durch die verschiedenen Zyklen auftretenden Koordinationsprobleme sei lediglich hingewiesen. Lösungen hierfür sind nicht Gegenstand der angestellten Untersuchungen. Es sei jedoch vermerkt, daß die Lösungen nicht ein allgemeines Konzept, wie etwa das der P- und V-Operationen (vgl. [DIJ 68]) verlangen. Im Bereich der Prozeßrechner-Software werden die "Grund-" und "Nebenzyklen" häufig durch Zeitberechnungen aufeinander abgestimmt.

Aus den Beschreibungen der Moduln läßt sich das in Fig. 2/15 dargestellte strukturelle Modulgeflecht ableiten. Man beachte das Auftreten verschiedener Knoten mit gleicher Markierung.

Figur 2/15 bietet nur eine grobe Darstellung der Modulstruktur des Programmsystems. Zur vollständigen Definition des Modulgeflechtes sind noch die Zugriffsoperation bzgl. der verwendeten D-Modul anzugeben. *)

*) Die E/A-Spezifikation der Zugriffsoperationen ist bei der Weiterführung des Beispiels in Kapitel 3 angegeben.

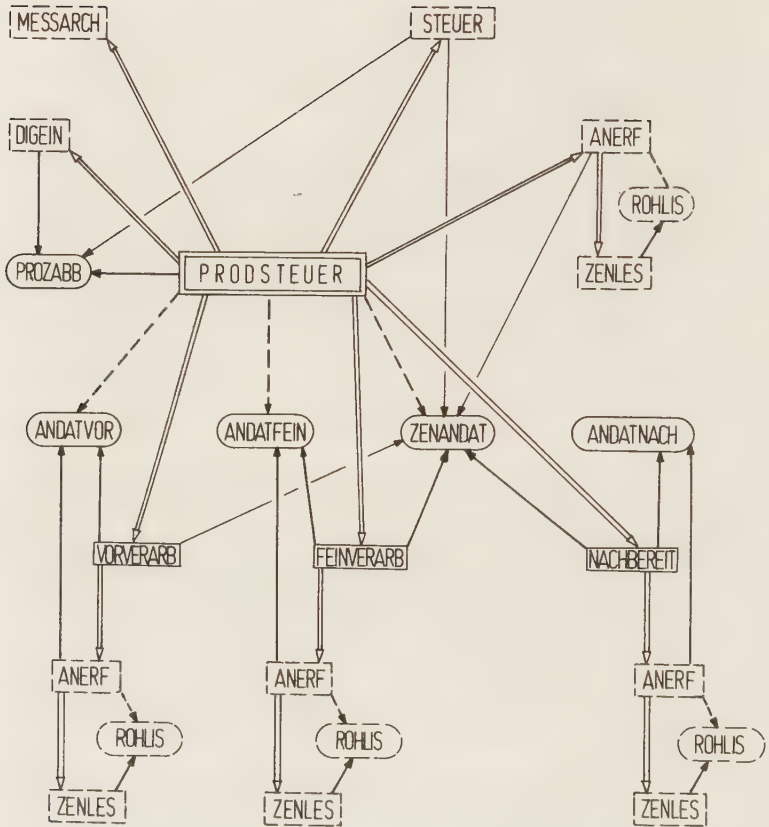


Fig. 2/15

- *Prozeßabbild*

• ENTRY	}	Liefern die Adressen der
• EXIT		binären Ein- bzw. Ausgänge
• READ_BIN	}	Bearbeitung der binären
• WRITE_BIN		Ein- bzw. Ausgänge
• GET_BIN	}	Dienen zur Darstellung von Ver-
• STORE_BIN		knüpfungen von Prozeßzuständen

- *zentrale Analogdaten*

• ENTRY		Liefert die Adressen der
		zentralen Analogeingänge
• PUT_ADR		Eintragen von "Hardware"-
		Adressen für die Analogeingänge
• GET_AN	}	Lesen und Schreiben von einge-
• STORE_AN		lesenen bzw. abgeleiteten (d.h.
		durch Verknüpfung entstandenen)
		Analogwerten

- *Analogdaten Vorverarbeitung, Analogdaten Feinverarbeitung, Analogdaten Nachbereitung*

Die Zugriffsoperationen sind wie beim D-Modul *zentrale Analogdaten*.

Man kann sich vorstellen, daß diesen vier D-Moduln jeweils der gleiche abstrakte Datentyp (vgl. Kapitel 3) zugrunde liegt.

- *Rohwertliste*

• WANT_READ		Trägt einen Lesewunsch bzgl.
		eines Analogeinganges ein.
• WHO_WANTS		Liefert die Art der Lese-
		wünsche zurück
• PUT_VALUE	}	Lesen und Schreiben von
• GET_VALUE		Rohwerten.

In Figur 2/16 und Figur 2/17 sind jeweils Darstellungen von Moduln aus Figur 2/15 angegeben.

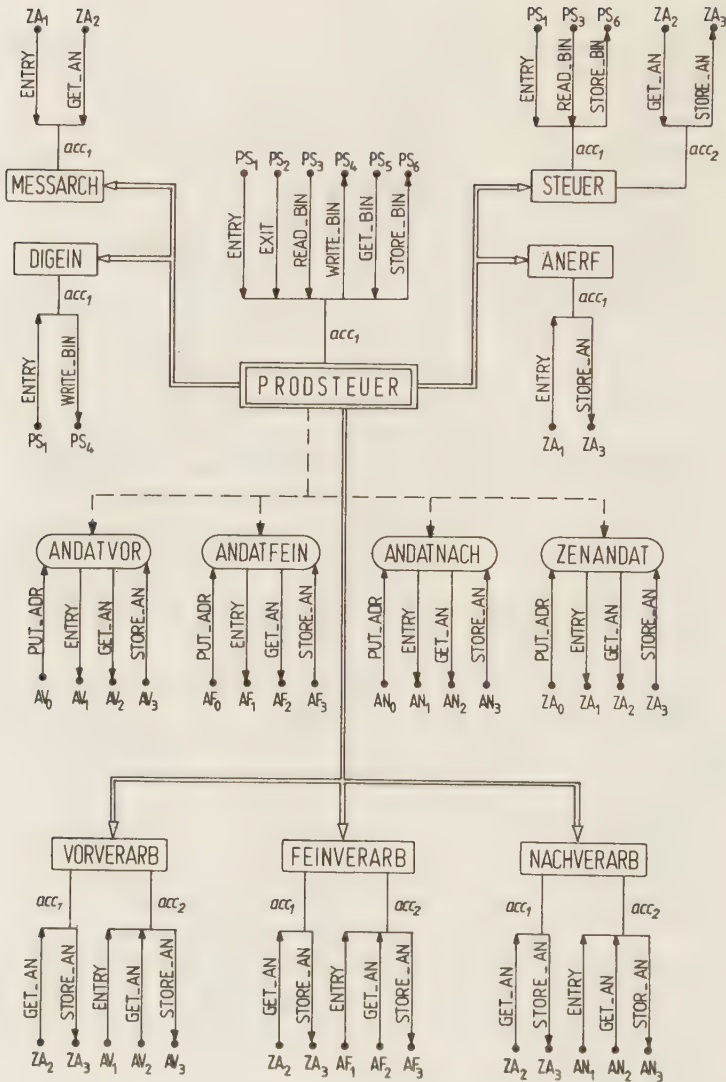


Fig. 2/16

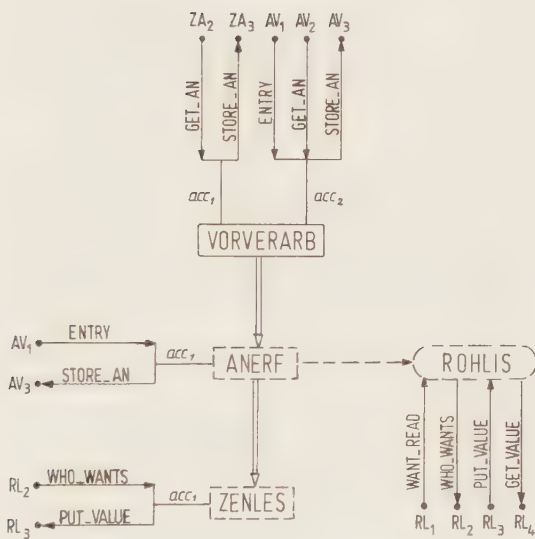


Fig. 2/17

Zum besseren Verständnis der Darstellungen in Fig. 2/16 bzw. 2/17 seien diese kurz erläutert. Betrachten wir zunächst Fig. 2/16.

- Durch die Definition des D-Moduls ZENANDAT in PRODSTEUER werden die Anschlußpunkte ZA_0 bis ZA_3 "defininiert".
- Betrachtet man die Schnittstellenbeschreibung des F-Moduls VORVERARB, so erkennt man, daß an diesen u.a. die Zugriffsrechte (ZENANDAT, GET_AN) ($\hat{=ZA_2}$) und (ZENANDAT, STORE_AN) ($\hat{=ZA_3}$) übertragen werden.
- Die Schnittstelle des (Haupt-)Moduls PRODSTEUER besteht aus Zugriffsrechten ($\hat{=PS_1, \dots, PS_6}$) zu einem *außerhalb des Systems* definierten D-Modul (PROZABB, vgl. Fig. 2/15). Diese Zugriffsrechte werden teilweise an den Modul STEUER übertragen (PS_1, PS_3, PS_6).

Figur 2/17 enthält eine gemeinsame Darstellung der F-Moduln VORVERARB und ANERF.

- Man erkennt die Kommunikation der beiden Moduln VORVERARB und ANERF über den D-Modul ANDATVOR: Übertragung der "Zugriffsrechte" AV_1 und AV_3 .
- Analog zu ANDATVOR wird der D-Modul ROHLIS gemeinsam von ANERF und ZENLES bearbeitet.

Beispiel 2.5

In diesem Beispiel soll eine einfache Problemstellung aus dem Bereich der Datenbank-Software, nämlich der Entwurf eines assoziativen Datenspeichers, behandelt werden. Die zugrunde liegende Terminologie ist aus [ENW 74] entnommen.

Problemstellung:

Der in [ENW 74] entworfene (pseudo-)assoziative Datenspeicher DATAS ist in Anlehnung an das in dieser Arbeit

vorgestellte Modul-Konzept neu zu strukturieren^{*)} und softwaremäßig zu implementieren. Die Benutzerschnittstelle bzw. die Zugriffsoperationen zu dem Speichersystem sind als Teil eines Dialogsystems vorgegeben. Die abspeicherbaren (vom Benutzer manipulierbaren) Datenelemente (Entities) sind:

- einfache Datenelemente
(Single-Value-Entities)
- Mengen
(Multi-Value-Entities)
- Relationen
(Relation-Entities)

Die Datenelemente können durch (Benutzer-)Namen adressiert werden, wobei für ein Element mehrere Namen (Synonyme) zugelassen sind. Die Zugriffsoperationen dienen der Manipulation von Einzeldaten, Mengen, Relationen und Namen, der Informationswiedergewinnung bzgl. dieser Objekte sowie der Initialisierung und Instandhaltung der Datenstrukturen.

Es ist davon auszugehen, daß die zu verwaltende Information wegen ihres Umfangs nicht ständig im Hauptspeicher gehalten werden kann.

Zur Implementierung des Systems werden folgende Entwurfsentscheidungen getroffen:

- Das (datenorientierte) Gesamtsystem wird als ein D-Modul ASSMEMORY mit den durch die Benutzerschnittstelle vorgegebenen Zugriffsoperationen realisiert.
- Die Zuordnung zwischen den (externen) Benutzernamen und den (internen) Datenelementen erfolgt in einem

^{*)} Diese Neustrukturierung ist Gegenstand einer Diplomarbeit [GAL 78] am Lehrstuhl für Programmiersprachen der Universität Erlangen-Nürnberg.

D-Modul NAMEMEMORY.

- Alle Datenelemente (im Falle der Relationen nur eine Beschreibung) werden in einem D-Modul DATAMEMORY abgelegt.
- Die zwischen Datenelementen möglichen Beziehungen (Relation-Entities) werden wegen der unterschiedlichen Verwendung von Relationen und der übrigen Elemente als sogenannte Triaden ((Relation, Element 1, Element 2)) in einem D-Modul TRIADEMEMORY gehalten.
- Die Verwaltung des (externen) Plattenspeichers bzw. die Zuordnung zwischen Extern- und Hauptspeicherseiten werden in einem D-Modul PAGETAB realisiert.
- Bei der Realisierung der einzelnen Zugriffsoperationen der Benutzerschnittstelle treten verschiedene Aktionen, die als mehrstellige Operationen bzgl. der D-Moduln NAMEMEMORY, DATAMEMORY, TRIADEMEMORY und PAGETAB aufgefaßt werden können, mehrfach auf. Diese Aktionen werden als F-Moduln realisiert.

Eine vollständige Beschreibung auf den jeweiligen D-Moduln definierten Zugriffsoperationen würde den Rahmen dieser Arbeit sprengen. Deshalb soll im folgenden nur eine Grobbeschreibung des Systems angegeben werden.

- Beschreibung von ASSMEMORY

Zur Manipulation der Objekte in ASSMEMORY dienen die folgenden (Benutzer-)Operationen:

MANIP_SINGLES	Manipulation von einfachen Datenelementen
MANIP_SETS	Manipulation von Mengen
MANIP_RELATIONS	Manipulation von Element-beziehungen

MANIP_NAMES	Manipulation von Namen
GET_INFO	Auslesen von Informationen über gespeicherte Objekte

- Beschreibung von NAMEMEMORY

Die Zugriffsoperationen STORE_NAME, DELETE_NAME und FIND_NAME dienen zum Abspeichern, Löschen bzw. Auffinden von Namen.

- Beschreibung von DATAMEMORY

Zum Eintragen, Löschen bzw. Auslesen der zu einem Element gehörigen charakteristischen Information (z.B. ob einfach, Menge oder Relation) sind die Operationen STORE_ENTITY_INFO, DELETE_ENTITY_INFO und GET_ENTITY_INFO definiert.

- Beschreibung von TRIADEMEMORY

Die Zugriffsoperationen sind:
STORE_TRIADE, DELETE_TRIADE, FIND_TRIADE

- Beschreibung von PAGETAB

Die einzige Operation PAGE bewirkt durch entsprechende Parametrisierung das Ein- bzw. Auslagern von Seiten sowie die Aktivierung von Hauptspeicherseiten.

- Als Beispiel für eine vierstellige Operation sei folgende Aktion beschrieben (F-Modul DELETE_ENTITY):

Zu löschen ist ein Datenelement, dessen Typ (einfach, Menge, Relation) zunächst nicht bekannt ist. Hierzu muß u.a. nachgefragt werden, ob es sich um eine Relation handelt (FIND_TRIADE). In jedem Fall ist die zu dem Element gehörige Information zu löschen (DELETE_ENTITY_INFO). Eventuell sind auch noch Synonyme aus der Datenstruktur zu löschen (DELETE_NAME). Da die zu lö-

schenden Informationen nicht unbedingt im Hauptspeicher verfügbar sein müssen, sind unter Umständen Plattenzugriffe nötig (PAGE).

Aus Figur 2/18 ist der Systementwurf ersichtlich. DELETE_ENTITY steht dabei stellvertretend für weitere F-Moduln.

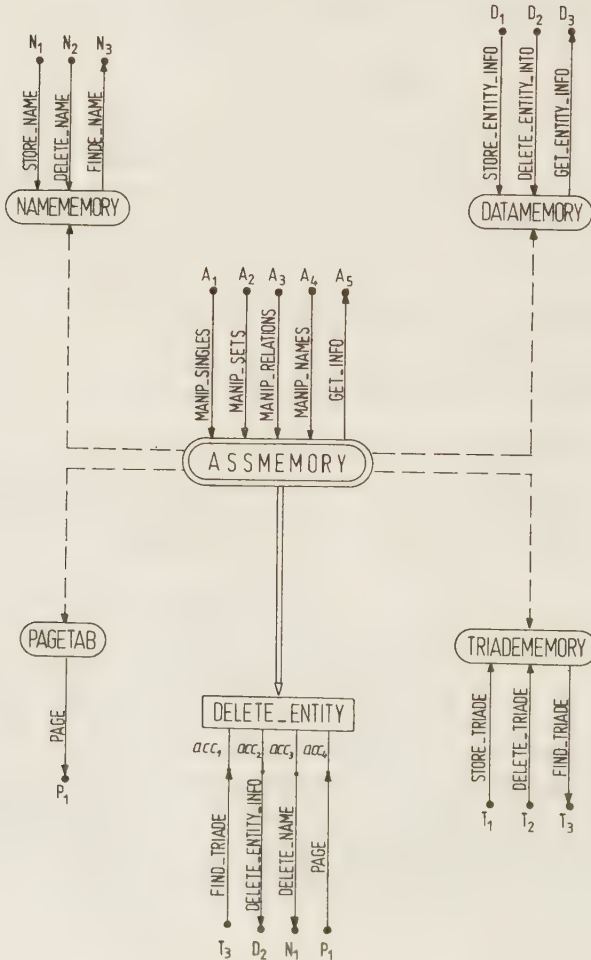


Fig. 2/18

KAPITEL 3 ENTWURF EINER PROGRAMMIERSPRACHLICHEN NOTATION

Nachdem in den Kapiteln 1 und 2 ein Modul-Konzept vorgestellt bzw. die statische Semantik eines Modulgeflechtes angegeben wurden, soll im folgenden eine programmiersprachliche Notation entworfen werden, die es gestattet, kontextunabhängige Programmmoduln, deren gegenseitige Beziehungen (schnittstellen) und Schutzaspekte von Objekten zu formulieren. Die Überprüfung von angegebenen Objektschutzbedingungen soll dabei bereits weitgehend zur Übersetzungszeit möglich sein. Es zeigt sich, daß diese Forderung ein hohes Maß an Redundanz mit sich bringt.

In diesem Kapitel soll *nicht* die vollständige Definition einer neuen Programmiersprache angegeben werden. Deshalb wird auf eine formale und vollständige syntaktische Beschreibung der im weiteren angegebenen Sprachkonstruktionen verzichtet. Diese sollen vielmehr anhand schematischer Darstellungen und mit Hilfe von Beispielen vorgestellt werden.

Ein Vergleich mit den in der Literatur bekannten Ansätzen zur programmiersprachlichen Beschreibung von (modularen) Programmsystemen wird in Kapitel 7 gegeben.

3.1 BEGRIFFSBILDUNG

Bevor in Kapitel 3.2 konkrete syntaktische Konstruktionen für Moduln angegeben werden, sei zunächst ein grober Überblick über die im Zusammenhang mit der programmiersprachlichen Notation verwendete Terminologie gegeben. Die Begriffe werden anhand von Beispielen in Kapitel 3.2 näher erläutert.

Ein *D-Modul* ist ein *Objekt*, dem ein bestimmter Speicherplatz (Repräsentierung) zugeordnet ist. Die Veränderung und Bearbeitung dieser Speicherzellen ist in eindeutiger Weise durch die einem D-Modul zugeordneten Zugriffsoperationen gegeben.*)

Der Aufbau der Repräsentierung und die einem D-Modul zugeordneten Operationen bestimmen dessen *Art*.

Das Konzept der *abstrakten Datentypen* dient zur Definition von D-Moduln mit strukturell gleicher**) Repräsentierung und gemeinsamen Zugriffsoperationen analogen E/A-Verhaltens***).

Im Vergleich zur Programmiersprache SIMULA 67 ([DMN 70]) kann festgestellt werden:

- Ein abstrakter Datentyp entspricht einer SIMULA-Klasse (class).
- Die mittels des Konstruktors new definierten Objekte (D-Moduln) sind von einer aus der Klasse (dem abstrakten Datentyp) abgeleiteten Art.

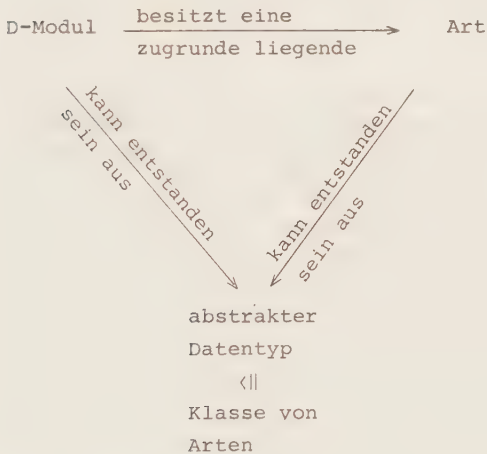
*) vgl. auch Definition 1.3, Kapitel 1, S.8.

**) Diese Begriffe werden nachfolgend an Beispielen erläutert.

***) Man vergleiche hierzu auch die Vorstellungen über "abstract types" in [PSW 76].

- Durch die Verwendung verschiedener Klassenparameter (Kreierungsparameter) können somit aus einer Klasse (einem abstrakten Datentyp) mehrere Arten abgeleitet werden.

Es besteht also folgender Zusammenhang:



Ein *F-Modul* ist eine prozedurähnliche Komponente eines Systems, deren Parameter sich nur aus Zugriffsrechten auf D-Moduln zusammensetzen. Der Grund für die Einschränkungen in bezug auf die Parameter liegt in der Festlegung, daß über Modulgrenzen hinweg nur geschützte, d.h. in D-Moduln gehaltene Daten, weitergereicht werden dürfen (vgl. auch Bemerkung nach Def. 1.6, S. 13).

Zur Erläuterung der Unterschiede der Begriffe D-Modul, abstrakter Datentyp und Art betrachte man nachfolgende Beispiele:

1. Eine Laufzeitkomponente zur Verwaltung einer einzigen, sequentiell bearbeitbaren Ausgabedatei kann als D-Modul aufgefaßt werden. Die zu bearbeitenden (geschützten) Daten setzen sich zusammen aus einer Dateibeschreibung, gewissen Datenpuffern und Pufferzeigern. Die durch den D-Modul definierten Zugriffsoperationen sind u.a. OPEN(..), CLOSE(..), OUTCHAR(..) und OUTLINE(..), die zum Öffnen bzw. Schließen der Datei und zur Ausgabe von Einzelzeichen bzw. Textzeilen dienen.

2. Die Verwendung verschiedenartiger blockstrukturierter Listen in Compilern legt die Implementierung eines abstrakten Datentyps "block structured list" nahe. Die zu bearbeitenden Datenstrukturen sind eine Liste von Informationseinheiten (z.B. Identifikatoren, Artindikatoren), eine Liste zur Darstellung der Blockstruktur und verschiedene Listenzeiger. Die benötigten Zugriffsoperationen sind u.a. INSERT_ELEM, IS_CONTAINED, BLOCK_BEGIN, BLOCK_END.
 Durch konkrete Angaben (Kreierungsparameter) bzgl. der zu verwaltenden Information, der maximalen Blockverschachtelung und der maximalen Anzahl von Listenelementen kann man aus dem abstrakten Datentyp block structured list verschiedene Arten ableiten.
 Man kann nun beispielsweise zwei D-Moduln M1 und M2 betrachten, die mit Hilfe des abstrakten Datentyps block structured list definiert werden.
 M1 sei ein D-Modul zur Verwaltung von Identifikatoren, M2 einer zur Verwaltung von Artindikatoren. Die maximale Blocktiefe sei jeweils gleich, die maximale Anzahl der Informationseinheiten nicht. Die M1 und M2 zugrunde liegenden Repräsentierungen können jedoch als strukturell gleich gelten, da sich die Felder zur Darstellung der Identifikatoren bzw. Artindikatoren

im wesentlichen nur durch die maximale obere Grenze unterscheiden. Analoges E/A-Verhalten der Zugriffsoperationen ist insofern gegeben, als beispielsweise die Operation `INSERT_ELEMENT` als Eingangsparameter jeweils ein Objekt der Art "Identifikator" bzw. "Art-indikator" (entsprechend der Zugehörigkeit zu M1 oder M2) zuläßt und bei Nichtüberschreitung der maximalen Anzahl der Elemente das Objekt in der entsprechenden Liste eingetragen wird, falls es dort noch nicht vorhanden ist.

Wir wollen uns nun im folgenden mit der Definition und Verwendung abstrakter Datentypen beschäftigen.

Ein abstrakter Datentyp wird definiert durch eine *Datentyp-Implementierung*. Hierbei handelt es sich um ein eigenständiges Programmsegment, das im folgenden als type module bezeichnet wird.

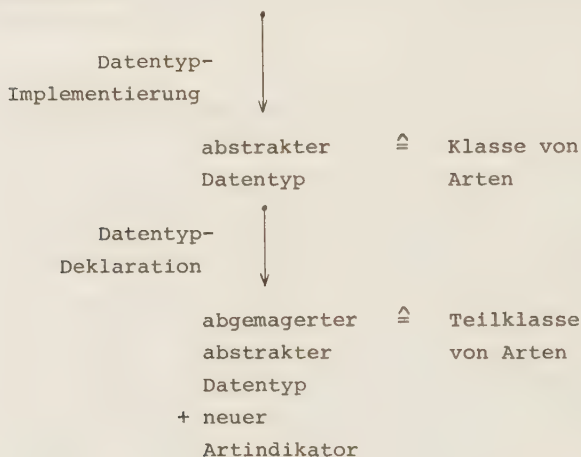
Neben der Definition der auf den Objekten des entsprechenden Typs zulässigen Operationen beinhaltet eine Datentyp-Implementierung eine Beschreibung der Repräsentierung. Um eine möglichst vielseitige Verwendung eines abstrakten Datentyps zu erreichen, kann man durch Kreierungsparameter Umfang und Art der Repräsentierung an die jeweiligen Anforderungen anpassen. Weiter ist vorgesehen, daß bei der Verwendung eines Typs bzgl. der bereitgestellten Zugriffsoperationen abgemagert werden kann.

Das Abmagerungskonzept verhindert, daß beim Binden der Moduln eines Geflechtes die nicht benötigten Zugriffsoperationen unnötig Speicherplatz belegen.

Durch eine *Datentyp-Deklaration* wird eine Teilklasse der durch einen abstrakten Datentyp definierten Klasse von Arten in einem Programmmodul bekannt gemacht. Dieser Teilklasse wird dabei ein neuer (abkürzenden) Artindikator

zugeordnet. Eine derartige Datentyp-Deklaration entspricht etwa einer Artdeklaration in ALGOL 68. In dieser Sprache kennt man zur Bildung von zusammengesetzten Arten u.a. die Strukturierung und die Reihung. Die Zugriffsmöglichkeiten auf Objekte der entsprechenden Art sind dabei implizit gegeben.

Zwischen den einzelnen Konstruktionen besteht folgender Zusammenhang:



Für die Definition eines abstrakten Datentyps durch einen type module gibt es vergleichbare Ansätze in der Literatur (vgl. z.B. class-Konzept in SIMULA 67 [DMN 70], cluster-Konzept [LIZ 74]). Das Konzept der Abmagerung abstrakter Datentypen geht über diese Möglichkeiten hinaus.

Nachfolgend soll nun beschrieben werden, welche Möglichkeiten zur Definition von D-Moduln vorgesehen sind.

Durch eine *D-Modul-Deklaration* wird der Name eines D-Moduls in einem Modul deklariert und dabei festgelegt, welche Zugriffsoperationen auf dem D-Modul verfügbar sind. Hierbei kann zwischen zwei Formen gewählt werden:

- Deklaration unter Verwendung eines abstrakten Datentyps.
- Deklaration unter Bezugnahme auf eine (textuell ausgelagerte) D-Modul-Implementierung (siehe nachfolgende Beschreibung).

Die zuerst genannte Form entspricht der in vielen Programmiersprachen bekannten Form der Deklaration eines Objektes durch Angabe der Art und des Bezeichners^{*)}.

Für die zweite Möglichkeit gibt es kein Analogon in den bekannten Programmiersprachen. Dieser Form der Objektdекlaration liegt folgender Mechanismus zugrunde:

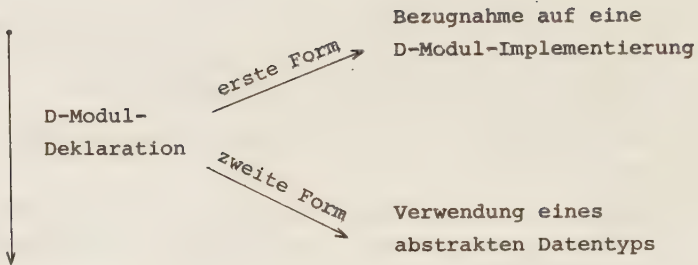
Durch eine *D-Modul-Implementierung* wird in einem eigenständigen Programmsegment (im weiteren data module genannt) der Aufbau einer Datenstruktur und die Definition der Zugriffsoperationen beschrieben und dem so geschaffenen Objekt ein Bezeichner zugeordnet.

Dieses Objekt kann nun durch eine Deklaration in genau einem Programmmodul bekannt gemacht werden.

Durch das textuelle Auslagern der Implementierung eines D-Moduls wird der Name dieses Objektes auch außerhalb seines Gültigkeitsbereiches sichtbar. In Kapitel 6 wird auf Probleme eingegangen, die sich hieraus ergeben.

Die Konstruktion soll durch das nachfolgende Diagramm erläutert werden.

^{*)} Beispiel in ALGOL 68: Sei amode eine Art, dann wird durch "amode X;" ein Objekt mit dem Namen X deklariert.



D-Modul wird in
einem Programmmodul
verfügbar

Im folgenden wird durch die Begriffe F-Modul-Implementierung und Untermodulspezifikation die Verwendung von F-Moduln erläutert.

Durch eine *F-Modul-Implementierung* wird in einem eigenständigen Programmsegment (im weiteren function module genannt) ein F-Modul und seine Internstruktur definiert und diesem ein Bezeichner zugeordnet.

Durch eine *Untermodul-Spezifikation* wird der Name eines F-Moduls in einem Modul bekannt gemacht und dabei festgelegt, welche Zugriffsrechte an ihn (aktuell) übertragen werden.

Neben diesen systembeschreibenden Sprachkonstruktionen werden noch solche zur Verwendung der Moduln benötigt:

- D-Modul-Zugriff
- F-Modul-Aufruf.

Ein *D-Modul-Zugriff* ist definiert durch Angabe des D-Modul-Bezeichners und der gewünschten Zugriffsoperation. Die

Operation ist dabei gegebenenfalls mit aktuellen Parametern zu versorgen. Ein *F-Modul-Aufruf* ist definiert durch Angabe des F-Modul-Bezeichners und der aktuellen D-Modul-Bezeichner als Parameter. Hierzu vergleiche man auch die Bemerkungen im Zusammenhang mit der Beschreibung der "interface"-Komponente von F-Moduln in Kap. 3.3 .

3.2 SPRACHKONSTRUKTIONEN IM ZUSAMMENHANG MIT D-MODULN

Wir betrachten zunächst die Sprachkonstruktionen zur Behandlung von abstrakten Datentypen und Arten.

Der prinzipielle syntaktische Aufbau einer Datentyp-Implementierung ist wie folgt^{*)}.

```

type module <Typ-Indikator>(<Kreierungsparameter>);
co Beschreibung der Schnittstelle nach außen oc
interface
    parameters <Beschreibung der Kreierungsparameter
                  in bezug auf Art und Eigenschaften>;
    public procedures
                  <Schnittstellenspezifikation
                  der Zugriffsoperationen>;
    errors      <Angabe der Fehlerausgänge>;
end interface;
co Internstruktur des Datentyps oc
realization
    representation
                  <Definition der zugrunde liegenden
                  Datenstruktur und deren Initialisierung>
    end representation;
    submodules <Spezifikation untergeordneter F-Moduln**) >
    end submodules;

```

^{*)} Syntaktische Variable sind in < > eingeschlossen.

^{**) vgl. Kapitel 3.3}

public procedures

<Realisierung der Zugriffsoperationen
in Abhängigkeit der Repräsentierung>;

local procedures

<Deklaration lokaler Prozeduren mit
globalem Zugriff auf die Repräsen-
tierung>;

end realization;

end type module <Typ-Indikator>;

□

Da abstrakte Datentypen zur Definition von D-Moduln verwendet werden, kann man den Aufbau einer Datentyp-Implementierung mit dem in Kapitel 1.3, Def. 1.6 angegebenen prinzipiellen Aufbau von D-Moduln vergleichen:

- Die Darstellung der zur Verfügung gestellten Schnittstelle nach außen ist durch die Komponenten "public procedures" und "errors" in der Schnittstellenbeschreibung gegeben.

Die "parameters"-Komponente stellt eine formale Beschreibung der von außen übertragenen Schnittstelle dar. Bei der Verwendung eines abstrakten Datentyps, z.B. in einer D-Modul-Deklaration, werden die formalen Parameter durch aktuelle ersetzt.

- Die Schnittstelle nach innen ist definiert durch die Komponente "submodules" und die in der Komponente "representation" deklarierten D-Moduln.

Beispiel 3.1: Definition eines abstrakten Datentyps
NICE_STACK durch eine Datentyp-Implementierung

```
type module  NICE_STACK (type EL_TYPE, int MAX_ST);
  interface
    parameters
      MAX_ST ≤ 100;

    public procedures
      POP, TOP returns EL_TYPE,
      PUSH (EL_TYPE ELEM), EMPTY returns bool

    errors OVERFLOW, UNDERFLOW
  end interface;

  realization
    representation
      [1 : MAX_ST] EL_TYPE STACK;
      int ST_POINTER := 0;

    end representation;

    public procedures
      POP : .... ;
      TOP : .... ;
      PUSH : ... ;
      EMPTY : .. ;

    end realization;
  end type module  NICE_STACK;
```

Es wird ein abstrakter Datentyp NICE_STACK mit zwei
Kreierungsparametern EL_TYPE und MAX_ST definiert. □

Bei der Syntax einer Datentyp-Deklaration ist nun darauf
zu achten, daß bei der Bezugnahme auf die Schnittstelle des
abstrakten Datentyps die Kreierungsparameter vollständig

angegeben werden, während bei den Zugriffsoperationen nur ein Teil referenziert werden muß (Abmagerungskonzept). Die Kreierungsparameter können formal oder aktuell referenziert werden. Formale Kreierungsparameter korrespondieren hierbei mit formalen Parametern des neu eingeführten Typs. Mit Hilfe der Datentyp-Implementierung des Beispiels 3.1 soll anhand eines Beispiels die Konstruktion erläutert werden.

Beispiel 3.2: Datentyp-Deklaration mit Einführung eines neuen Artindikators

```
abstrdata
    POOR_STACK (type T) =
        NICE_STACK (type T,4)
        with PUSH (T X), POP,
            errors OVERFLOW, UNDERFLOW;
end abstrdata;
```

Es wird ein Artindikator POOR_STACK mit Parameter T der Art type für Keller der Tiefe 4 deklariert. Auf D-Moduln der Art POOR_STACK kann nur mit den Operationen PUSH(..) und POP zugegriffen werden. □

Während man bei den Zugriffsoperationen im Prinzip beliebig abmagern kann, ist bei den Fehlerausgängen Vorsicht geboten: Die Aktivierung der Fehlerausgänge geschieht bei Ausführung der Zugriffsoperationen. Somit sind alle diejenigen Ausgänge hinzuzunehmen, die von den angegebenen Zugriffsoperationen aktiviert werden können. In Beispiel 3.2 wäre es sinnlos, einen der beiden Fehlerausgänge OVERFLOW bzw. UNDERFLOW wegzulassen.

Im folgenden werden nun Sprachkonstruktionen zur Behandlung von D-Moduln betrachtet.

Der syntaktische Aufbau einer D-Modul-Implementierung ist analog zum Aufbau einer Datentyp-Implementierung.

Die von außen übertragene Schnittstelle setzt sich aus Externbezügen zu anderen D-Moduln (zum Aufbau von Netzen von D-Moduln) und Ladezeit-Konstanten als Kreierungsparameter zusammen *).

Bezogen auf das Graph-Modell eines modularen Programmsystems entsprechen derartige Externbezüge Kanten e mit $art(e) = acc$. Es sind für diese Externbezüge somit die Bedingungen der Definitionen 2.4(2.) und 2.6 bzgl. der Verteilung von Zugriffsrechten zu beachten.

In Anlehnung an Beispiel 3.1 ist in Beispiel 3.3 die Implementierung eines D-Moduls KELLER angegeben.

Beispiel 3.3: D-Modul-Implementierung

```
data module KELLER;

  interface

    public procedures
      POP, TOP returns struct (int A, string B),
      PUSH (struct (int A, string B)),
      EMPTY returns bool;

    errors OVERFLOW, UNDERFLOW

  end interface;
```

*) Zur Erinnerung:

Bei D-Moduln unterscheiden wir zwei Arten von Parametern:

- Kreierungsparameter dienen zur Definition der Repräsentierung
- Externbezüge zu anderen D-Moduln entsprechen den Parametern bei F-Moduln, die übertragene Zugriffsrechte darstellen.

```

realization
  representation
    [1 : 100] struct (int A, string B) STACK;
    int ST_POINTER := 0
  end representation;
  public procedures
    POP :    ... ;
    TOP :    ... ;
    PUSH :   ... ;
    EMPTY :  ... ;
  end realization;
end data module KELLER;

```

□

Bei einer D-Modul-Deklaration unter Bezugnahme auf die Schnittstelle eines abstrakten Datentyps können folgende Fälle unterschieden werden:

- Die D-Modul-Deklaration bezieht sich auf eine innerhalb desselben Moduls erfolgte Datentyp-Deklaration.
- Die D-Modul-Deklaration bezieht sich direkt auf die Schnittstelle eines abstrakten Datentyps.

In beiden Fällen müssen alle formalen Parameter durch aktuelle ersetzt werden.

- Eine weitere Möglichkeit einer D-Modul-Deklaration besteht durch vollständige Bezugnahme auf die Schnittstelle einer D-Modul-Implementierung.

In den Beispielen 3.4, 3.5 und 3.6 ist unter Verwendung der Beispiele 3.1, 3.2 bzw. 3.3 jeweils eine der drei möglichen Formen von D-Modul-Deklarationen angegeben.

Beispiel 3.4

```
declare POOR_STACK(int) KELLER1;
```

Unter Berücksichtigung der Beispiele 3.2 und 3.1 erkennt man, daß durch obige Deklaration ein D-Modul mit dem Bezeichner KELLER1 eingeführt wird. Dem D-Modul liegt die Datenstruktur

```
[1 : 4] int STACK;  
int ST_POINTER := 0;
```

zugrunde.

Als Zugriffsoperationen sind zugelassen PUSH(...) und POP. Als Fehlerausgänge stehen UNDERFLOW und OVERFLOW zur Verfügung.

□

Beispiel 3.5

Die gleiche Wirkung wie in Beispiel 3.4 wird durch die Deklaration

```
declare NICE_STACK (int,4) KELLER1  
    with PUSH (int), POP,  
        errors OVERFLOW, UNDERFLOW;
```

erzielt.

□

Beispiel 3.6

```
include KELLER  
  
    with POP, TOP returns struct (int A, string B),  
        PUSH (struct (int A, string B)),  
        EMPTY returns bool,  
        errors OVERFLOW, UNDERFLOW;
```

Der D-Modul mit der Bezeichnung KELLER wird unter vollständiger Bezugnahme auf die Schnittstelle der D-Modul-

Implementierung aus Beispiel 3.3 in einem Modul bekannt gemacht. Für diese Form der Deklaration wurde das Wortsymbol include gewählt, da das wesentliche in diesem Fall das "Einschleusen" einer textuell ausgelagerten Definition eines D-Moduls ist.

□

In Beispiel 3.7 werden die Verwendung von *Fehlerausgängen* und der *Zugriff* auf D-Moduln demonstriert. Die syntaktische Notation ist wie folgt:

```
<Fehlerausgang> . <D-Modul-Name> bzw.
<Operationsbezeichnung
+ aktuelle Parameter> . <D-Modul-Name>
```

Die Anweisung "on ...;" soll dabei wie die in PL/1 bekannten ON-Bedingungen wirken. Der D-Modul KELLER sei wie in Beispiel 3.6 deklariert.

Beispiel 3.7

```
declare KELLER ...;
.
.
.
on OVERFLOW.KELLER : exit;
.
.
.
while ...
    begin
        .
        .
        PUSH(..).KELLER;
        .
        .
    end;
```

□

Zur Verwendung der Fehlerausgänge vergleiche man auch die Bemerkungen in Kapitel 2.5.

3.3 SPRACHKONSTRUKTIONEN IM ZUSAMMENHANG MIT F-MODULN

Wir betrachten zunächst die syntaktische Konstruktion für eine F-Modul-Implementierung.

Der prinzipielle Aufbau ist wie folgt:

```

function module <Bezeichner>(<Param.-Liste>);
  co Schnittstelle nach außen oc

  interface <Beschreibung von formalen Zugriffs-
              rechten, die bei einem Aufruf des
              F-Moduls aktualisiert werden>

end interface;

realization
co Internstruktur oc

  data

    provides data
      <Deklaration von D-Moduln, deren Bear-
        beitung an untergeordnete F-Moduln
        weitergegeben werden kann>;

    local data
      <Deklaration lokaler Datenstrukturen,
        deren Bearbeitung nicht an andere Mo-
        dule weitergereicht werden kann>

  end data;

  submodules
    <Spezifikation untergeordneter F-Moduln;
    dabei werden Kommunikationswege der F-Moduln untereinander sichtbar gemacht>

  end submodules;

```

```

algorithm
  begin
    <Deklaration lokaler Prozeduren>;
    <Formulierung des dem F-Modul zu-
      grunde liegenden Algorithmus>
  end
end realization
end function module <Bezeichner>;

```

□

Aus der für F-Moduln möglichen Form der Schnittstelle und aus der Unterscheidung der in F-Moduln definierten Daten in "provides data" und "local data" kann man folgendes Datenkonzept erkennen:

- Es gibt Daten, die an andere Moduln weitergereicht werden dürfen und solche, die ausschließlich lokal verwendet werden.
- Daten, die weitergereicht werden dürfen, sind stets geschützte Daten, d.h. in D-Moduln "verpackt" ("gekapselt", "versteckt").

Durch die in der "provides data"-Komponente deklarierten D-Moduln werden - bezogen auf das Graphmodell aus Kapitel 2 - Kanten mit der Markierung *def* eingeführt. Durch Untermodulspezifikationen werden Kanten der Art *dom* und durch die Übertragung von Zugriffsrechten Kanten der Art *acc* eingeführt. Es sind für diese Modulvernetzungen somit die Bedingungen der Definitionen 2.2, 2.3 und 2.5 zu beachten.

Das Prinzip, daß jeder Bezug zwischen Moduln aus einer Schnittstellenbeschreibung ersichtlich ist, hat zur Folge, daß die Bedingung von Definition 2.4 ("information hiding") automatisch erfüllt ist.

Die Aussage von Satz 2.3 wird erfüllt, wenn man verlangt, daß die an Untermoduln übertragenen Zugriffsrechte sich nur auf D-Moduln aus der "interface"-Komponente (formale D-Moduln) bzw. der "provides data"-Komponente beziehen.

Zwischen dem hier beschriebenen syntaktischen Aufbau und dem in Kapitel 1.3, Def. 1.5 angegebenen prinzipiellen Aufbau von F-Moduln besteht folgender Zusammenhang:

- Die Schnittstelle nach außen entspricht der Komponente "interface".
- Die Schnittstelle nach innen setzt sich aus den Komponenten "provides data" und "submodules" zusammen.

Bevor im nachfolgenden Abschnitt 3.4 Beispiele für die programmiersprachliche Notation angegeben werden, soll kurz auf die Form der Untermodulspezifikation, die Beschreibung der übertragenen Zugriffsrechte (Schnittstelle nach außen), die Syntax eines F-Modul-Aufrufs und eine Möglichkeit der E/A-Spezifikation von Zugriffsoperationen eingegangen werden.

- Untermodulspezifikation

In Kapitel 1.3 wurde verlangt, daß bei der Darstellung der Informationskanäle zwischen untergeordneten F-Moduln alle dynamisch möglichen Bezüge zu bereitgestellten D-Moduln sichtbar werden müssen. Dies kann dazu führen, daß aktuell auf einer Parameterposition die Zugriffsberechtigungen zu mehreren D-Moduln beschrieben werden.

Die allgemeine Form einer Untermodulspezifikation ist wie folgt:

<Untermodul-Bezeichner>

access to <Liste von Parameterbeschreibungen>

end access;

Eine Parameterbeschreibung hat die Form:

```
<Liste von aktuellen D-Modul-Bezeichnern>
  with <Liste von aktuellen E/A-Spezifikationen
        von Zugriffsoperationen>;
```

Formal entspricht eine Untermodulspezifikation der Darstellung der an einen F-Modul übertragenen Zugriffsrechte, zusammen mit einer Parametrisierung (vgl. Kap. 2.2, Def. 2.13 bzw. 2.16).

Beispiel 3.8

Es ist die Spezifikation eines Untermoduls X anzugeben. Dynamisch mögliche Bezüge auf einer Parameterposition bestehen zu den D-Moduln D1 und D2, jeweils mit den Zugriffsoperationen OP1, OP2.

Es ergibt sich folgende Darstellung:

```
submodules
.
.
X access to
    D1, D2 with OP1 <aktuelle E/A-Spezifikation>,
              OP2 <aktuelle E/A-Spezifikation>;
end access;
.
.
end submodules;
```

□

- Beschreibung übertragener Zugriffsrechte (Schnittstelle nach außen)

Die Untermodulspezifikation mit aktuellen D-Modul-Bezeichnern entspricht der Beschreibung der übertragenen Zugriffsrechte (in der "interface"-Komponente) mit formalen D-Modul-Bezeichnern.

Eine Parameterbeschreibung in der Schnittstelle nach außen wird wie folgt notiert:

<formaler D-Modul-Bezeichner>

with <Liste von formalen E/A-Spezifikationen von Zugriffsoperationen>;

Beispiel 3.9

Die Schnittstellenbeschreibung des F-Moduls X aus Beispiel 3.8 hat somit folgendes Aussehen (DF ist ein formaler D-Modul-Bezeichner!):

function module X (DF);

interface

DF with OP1 <formale E/A-Spezifikationen>,
OP2 <formale E/A-Spezifikationen>;

end interface;

□

Es ist zu beachten, daß die Bezeichnungen der Operationen (OP1, OP2) bzgl. des formalen D-Moduls DF mit denen bzgl. der entsprechenden aktuellen D-Moduln (D1, D2; vgl. Beispiel 3.8) übereinstimmen.

Ein etwas komplizierterer Parametermechanismus würde entstehen, wenn man zuläßt, daß in der "interface"-Komponente eines F-Moduls neben den formalen D-Modul-Namen auch formale Operationsnamen auftreten. Auf diese Möglichkeit sei jedoch lediglich hingewiesen. Wir gehen im folgenden von der zuerst dargestellten Alternative aus.

- F-Modul-Aufruf

Beim Aufruf eines F-Moduls wird dieser mit aktuellen D-Modul-Namen versorgt.

Es ergibt sich folgende Form:

call <F-Modul-Bezeichner>(<Liste aktueller D-Modul-Namen>);

Die Menge aller dynamisch möglichen Aufrufe eines F-Moduls ergibt sich bereits aus der entsprechenden Untermodulspezifikation. Die formale Korrektheit eines entsprechenden Aufrufs ist daher bereits zur Übersetzungszeit überprüfbar.

Für den F-Modul X aus Beispiel 3.8 sind somit zulässig:

- call X (D1);
- call X (D2);

- E/A-Spezifikation von Zugriffsoperationen

Die E/A-Spezifikation setzt sich zusammen aus der Beschreibung von Eigenschaften der Eingangs- und Ausgangsparameter. Die Beschreibung an sich kann wiederum parametrisiert sein.

Die allgemeine Form einer E/A-Spezifikation ist wie folgt:

<Bezeichner der Operation>

(<Liste der Eing.-Parameter>; <Parameterliste für die
Beschreibung der Eing.-
Parameter>

where <Beschreibung von Eigenschaften der
Eingangs-Parameter>)

{returns (<analoge Beschreibung für die
Ausgangsparameter>) /

returns <Ergebnisart bei Operationen, die die
Form einer Funktionsprozedur haben>*)

*) Die Notation {Alternative1/Alternative2} bedeutet, daß höchstens eine der beiden Alternativen gewählt werden kann.

Aktuelle und formale E/A-Spezifikation (in einer Untermodulspezifikation bzw. in der "interface"-Komponente eines F-Moduls) unterscheiden sich durch formale bzw. aktuelle Parameterlisten für die Beschreibung. Der Unterschied zwischen beiden Formen kann aus Beispiel 3.10 im folgenden Abschnitt 3.4 anhand der Zugriffsoperationen zum D-Modul PROZABB entnommen werden.

3.4 BEISPIELE

Zur Demonstration der programmiersprachlichen Notation werden im folgenden die beiden Beispiele aus Kapitel 2.4 dargestellt.

Beispiel 3.10

Betrachten wir zunächst das Beispiel aus der Prozeßrechner-Software (Beispiel 2.4, S. 59). Eine programmiersprachliche Darstellung des Haupt-Moduls PRODSTEUER erhält man, indem die Darstellung aus Figur 2/16 (in Kapitel 2.4, S. 66) entsprechend übertragen wird. Die Schnittstellenbeschreibungen der Moduln werden dabei insofern verfeinert, als aus der programmiersprachlichen Notation nicht nur (wie bereits aus Figur 2/16 ersichtlich) die Verteilung der Zugriffsrechte und die Modulstruktur hervorgeht, sondern auch das E/A-Verhalten der Zugriffsoperationen erkennbar wird.

Auf den folgenden Seiten wird eine Darstellung des F-Moduls PRODSTEUER mit anschließenden Erläuterungen gegeben.

① main function module PRODSTEUER(PROZABB);

interface co Beschreibung der Kommunikation
mit dem zu steuernden Prozeß oc

② PROZABB with

ENTRY (int I; int K where $1 \leq I \leq K$)
returns bin addr,

EXIT (int I; int L where $1 \leq I \leq L$)
returns bin addr,

co Adressen binärer Ein- und Ausgänge oc

READ_BIN(bin addr) returns bit,

WRITE_BIN(bin addr, bit),

co Bearbeitung der binären Ein- und
Ausgänge oc

GET_BIN (int I; int M where $1 \leq I \leq M$)
returns bit,

STORE_BIN (int I, bit; int N where $1 \leq I \leq N$);

co Darstellung von Verknüpfungen von Pro-
zeßzuständen oc

end interface;

co Internstruktur von PRODSTEUER oc

realization

data

provides data

3

declare ANALOG (10,15) ZENANDAT with

PUT_ADR (int I, an addr; 10 where $1 \leq I \leq 10$),

co Eintragen von "Analog-Adressen" (an addr) oc

ENTRY (int I; 10 where $1 \leq I \leq 10$)

returns an addr,

co Auslesen von "Analog-Adressen" oc

GET_AN (int) returns real,

STORE_AN (int, real);

co Lesen und Schreiben von Analogwerten oc

declare ANALOG(...)ANDATVOR with;

declare ANALOG(...)ANDATFEIN with ...;

declare ANALOG(...)ANDATNACH with ...;

end data;

submodules

co Auswahl aus den in Figur 2/16 dargestellten Untermoduln von PRODSTEUER oc

4

VORVERARB

access toZENANDAT withSTORE_AN(C1, real),GET_AN(C2) returns bool;ANDATVOR with

ENTRY(...),

GET_AN(...),

STORE_AN(...);

end access;

.

STEUER

access toPROZABB withENTRY (int I; C3 where $1 \leq I \leq C3$)returns bin addr,READ_BIN (bin addr) returns bitSTORE_BIN (bin addr, bit);ZENANDAT withGET_AN (int) returns real,STORE-AN (int, real);end access

.

end submodules;

.

.

end realization;end function module PRODSTEUER;

Erläuterungen

- ① main ist eine Kennzeichnung für den Hauptmodul (Funktion des Knotens s in Def. 2.1, S. 24).

- ② PROZABB ist ein formaler D-Modul-Name

Die formale E/A-Spezifikation von ENTRY setzt sich zusammen aus:

- Parameterliste: int I
- (formale) Par.-Liste (int K)
für die Beschreibung der Eingangsparameter
(where $1 \leq I \leq K$)
- Ergebnisart: bin addr

- ③ ZENANDAT ist ein aktueller D-Modul-Name

Es handelt sich um eine D-Modul-Deklaration unter Verwendung der Schnittstelle eines abstrakten Datentyps ANALOG mit den Kreierungsparametern 10 und 15.

- ④ Die Untermodulspezifikation für VORVERARB beinhaltet u.a. eine Einschränkung der Zugriffsberechtigung bzgl. des D-Moduls ZENANDAT: Für VORVERARB sind nur STORE_AN und GET_AN zulässig. Die (aktuelle) E/A-Spezifikation STORE_AN (C1, real) bedeutet: Bei (aktuellem) Zugriff mit STORE_AN muß der erste Parameter stets die Konstante C1 sein. Bei der (formalen) E/A-Spezifikation von STORE_AN in der "interface"-Komponente der nachfolgenden Beschreibung von VORVERARB durch

STORE_AN (int K1, real; where is-const(K1))
entspricht K1 der Konstanten C1.

Das Zusammenspiel zwischen (aktueller) Untermodulspezifikation und der (formalen) Beschreibung der "interface"-Komponente soll im folgenden am Beispiel des F-Moduls VORVERARB demonstriert werden.

```

function module VORVERARB(ZAD,ADV);
  interface
    ZAD with
      STORE_AN (int K1, real; where is-const(K1)),
      GET_AN (int K2; where is-const(K2))
        returns real;
    ADV with
      ENTRY(...), GET_AN(...), STORE-AN(...);
  end interface;
  .
  .
end function module VORVERARB;

```

□

Beispiel 3.11

Bei der programmiersprachlichen Darstellung des Haupt-Moduls ASSMEMORY aus Beispiel 2.5, S. 68 wird auf eine Beschreibung des E/A-Verhaltens der einzelnen Zugriffsoperationen verzichtet. Die Übertragung von Fig. 2/18 in die programmiersprachliche Notation spiegelt lediglich die Struktur wider.

```

main data module ASSMEMORY (<Konst.-Liste>);
  interface
    parameters <Beschreibung der Ladezeit-Konstanten
      von <Konst.-Liste>>;
    public procedures
      MANIP-SINGLES(...),
      MANIP_SETS(...),
      MANIP_RELATIONS(...),
      MANIP_NAMES(...),
      GET_INFO(...),
  end interface;

```

co Internstruktur von ASSMEMORY oc

realization

representation

include

PAGETAB(...)

with PAGE(...);

NAMEMEMORY(...)

with STORE_NAME(...),

DELETE_NAME(...),

FIND-NAME(...);

DATAMEMORY(...)

with STORE_ENTITY_INFO(...),

DELETE_ENTITY_INFO(...),

GET_ENTITY_INFO(...);

TRIADEMEMORY(...)

with STORE_TRIADE(...),

DELETE_TRIADE(...),

FIND_TRIADE(...);

<Deklaration weiterer lokaler Daten>

end representation;

```

submodules
.
.
.
DELETE_ENTITY

    access to

        TRIADE_MEMORY
        with FIND_TRIADE(...);

        DATAMEMORY
        with DELETE_ENTITY_INFO(...);

        NAME_MEMORY
        with DELETE_NAME;

        PAGETAB
        with PAGE(...);

    end access;
.
.
.
end submodules;
public procedures
    MANIP_SINGLES: ...;
    .
    .
    .
    GET_INFO: ...;

end realization

end data module ASSMEMORY;

```

□

KAPITEL 4 IMPLEMENTIERUNGSASPEKTE

4.1 ZIELSETZUNG

In diesem Kapitel werden Problemstellungen behandelt, die sich ergeben, wenn man an die Implementierung eines Compilers für die in Kap. 3.2 und 3.3 entworfenen Sprachkonstruktionen denkt.

Zunächst werden dabei Listenstrukturen betrachtet, die man zur Übersetzungszeit zur Beschreibung der Modulstrukturen benötigt. Ein weiterer Gegenstand der Untersuchungen ist die Speicherstruktur eines modularen Programmsystems zur Laufzeit. Als generelles Grundkonzept für die Implementierung steht die strikte Trennung der Schnittstelle nach außen von der Internstruktur im Vordergrund.

4.2 BESCHREIBUNG VON PROBLEMEN ZUR ÜBERSETZUNGSZEIT

Um zur Übersetzungszeit die korrekte Verwendung von Programmmoduln prüfen zu können, werden Listenstrukturen benötigt, die eine Beschreibung der einzelnen Moduln und deren gegenseitige Vernetzung beinhalten. Im Prinzip ergibt sich hierbei eine Codierung des in Kapitel 2 definierten Graphmodells in einer Listenstruktur. Durch die programmiersprachliche Notation kommen die Berücksichtigung formaler und aktueller Parameter von Moduln und deren Beschreibung hinzu.

Die im folgenden angegebenen Listen können als verallgemeinerte Symbollisten betrachtet werden, wie sie etwa in [SCH 75a] angegeben sind. Im Anschluß an eine detaillierte Beschreibung des Aufbaus der Listen wird anhand eines abstrakten Beispiels deren Verwendung erläutert.

Zur Beschreibung einer Modulstruktur definieren wir drei Datenstrukturen:

- Liste zur Beschreibung der F-Moduln (*FMODBESCHR*)

Die Verkettungen innerhalb dieser Liste spiegeln dabei die F-Modul-Hierarchie wider (vgl. r_{FF} in Def. 2.1, S. 24). Die einzelnen Elemente der Liste stellen eine Beschreibung von F-Moduln auf der Grundlage von Def. 2.15, S. 47 dar.

- Liste zur Beschreibung der D-Moduln (*DMODBESCHR*)

Aus den Verkettungen innerhalb dieser Liste ist die D-Modul-Hierarchie ableitbar (vgl. r_{DD} in Def. 2.4, S. 33).

Die einzelnen Elemente der Liste beinhalten eine Darstellung von D-Moduln auf der Grundlage von Def. 2.14, S. 46.

- Liste zur Darstellung der an die Moduln übertragenen Zugriffsrechte*) (BESCHRÜZR)

Diese Liste beschreibt im wesentlichen die zwischen den Moduln definierten Zugriffskanten.

Modulverbindungen, die aufgrund der "Schnittstellen nach innen" existieren (Definitions- und Untermodulkanten), werden direkt als Verweise zwischen Elementen der Datenstrukturen FMOBESCHR und DMOBESCHR dargestellt.

Faßt man die Tupel zur Beschreibung von D- bzw. F-Moduln in den Definitionen 2.14 bzw. 2.15 als "structures" (im Sinne von ALGOL 68) mit Komponenten variabler Größe auf, so erkennt man, daß bei der Implementierung der Listenelemente von FMOBESCHR und DMOBESCHR die Leitzellentechnik verwendet werden kann.

Im folgenden wird jeweils der Aufbau der Datenstrukturen und zugehöriger Unterlisten im Detail angegeben.

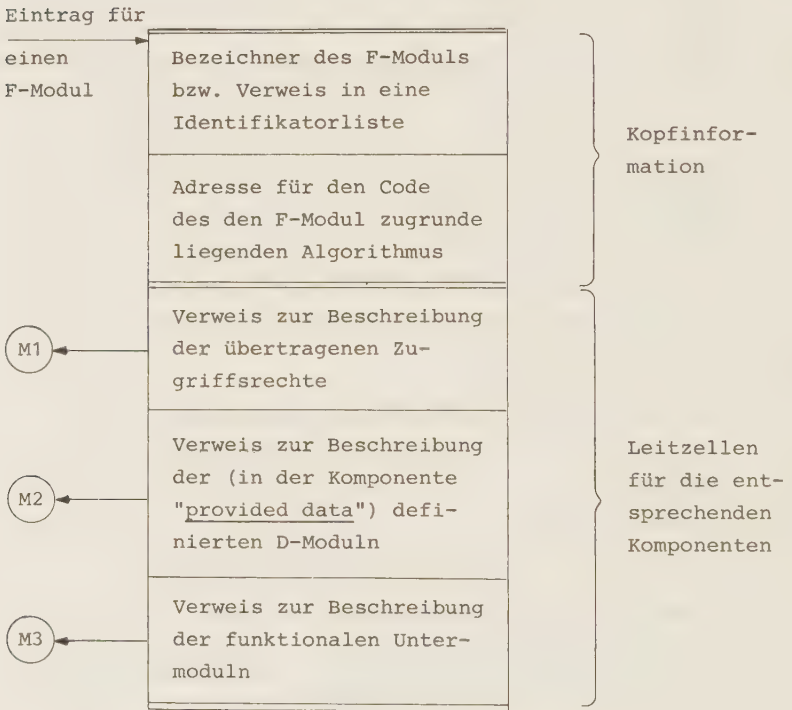
- Aufbau der Liste der F-Moduln

Der Beschreibung von F-Moduln gemäß Def. 2.15, S. 47 folgend, enthält ein Element von FMOBESCHR folgende Information:

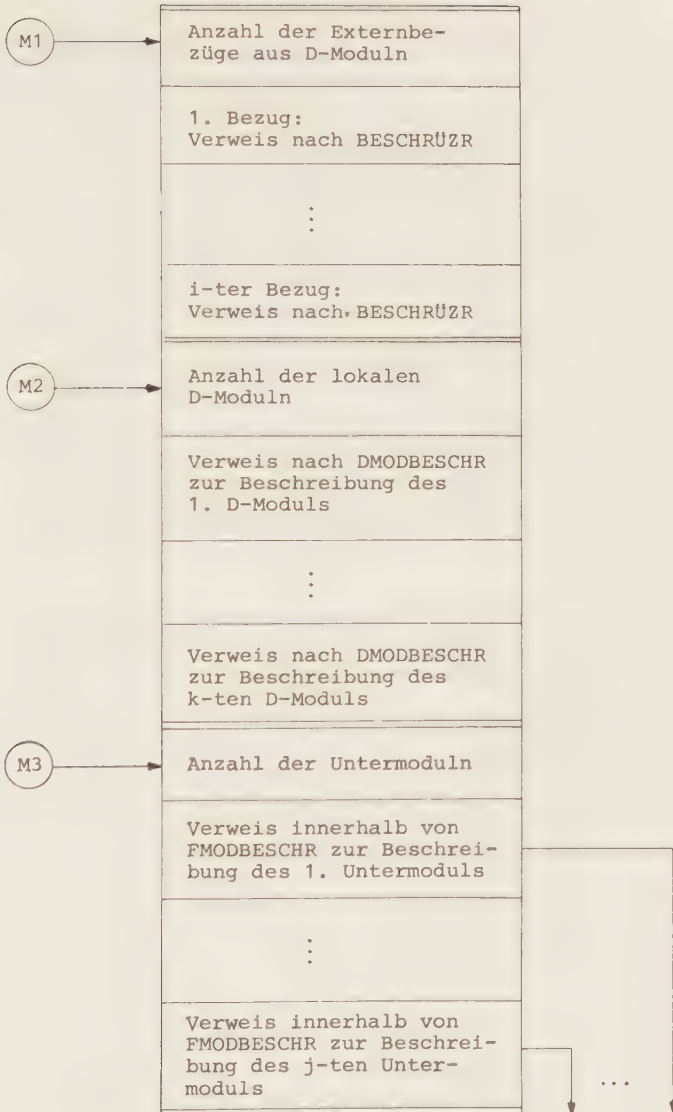
- F-Modul-Identifikation
- Beschreibung der übertragenen Zugriffsrechte
- Beschreibung der funktionalen Untermoduln
- Beschreibung der innerhalb des jeweiligen F-Moduls definierten (lokalen) D-Moduln.

*) vgl. hierzu ÜZR(k) in Def. 2.13

Die Darstellung dieser Information, unter Zuhilfenahme der Leitzellentechnik, ergibt für ein Element von FMOBESCHR den durch Fig. 4/1a und 4/1b gegebenen Aufbau.



Figur 4/1a



Figur 4/1b

Der Verweis M3 in Fig. 4/1a kann eingespart werden, wenn man verlangt, daß ein Element von FMOBESCHR zusammenhängend gespeichert wird. In diesem Fall kann die Beschreibung der funktionalen Untermoduln direkt im Anschluß an die "Zelle" abgelegt werden, die den Verweis M2 enthält. Der hier vorgestellte Aufbau wird dennoch bevorzugt, da er flexibler ist (nur die Leitzellen und die Kopfinformation müssen zusammenhängend sein).

Tritt nun während der Übersetzung des einem F-Modul zugrunde liegenden Algorithmus (vgl. Kap. 3.3, S. 90, Aufbau einer F-Modul-Implementierung) der Aufruf eines F-Moduls bzw. der Zugriff zu einem D-Modul auf, so können mit Hilfe der Datenstruktur FMOBESCHR jeweils die Zulässigkeit und die formale Korrektheit dieser Aktionen geprüft werden.

- Aufbau der Liste der D-Moduln

Die Elemente der Datenstruktur DMOBESCHR enthalten folgende Information:

- D-Modul-Identifikation
- Beschreibung der Kreierungsparameter
- Beschreibung übertragener Zugriffsrechte
- Beschreibung der Zugriffsoperationen
- Beschreibung funktionaler Untermoduln
- Beschreibung der innerhalb des jeweiligen D-Moduls definierten (lokalen) D-Moduln

Der Datenstruktur DMOBESCHR sind zwei Hilfslisten (*PARBESCHR*, *BESCHROP*) angegliedert. Die erste Liste (*PARBESCHR*, Liste der Parameterbeschreibungen) enthält jeweils die Beschreibung der Kreierungsparameter und der

Parameter der Zugriffsoperationen eines D-Moduls. In der zweiten Liste (BESCHROP, Beschreibung der Operationen) werden jeweils alle zu einem D-Modul gehörigen Zugriffsoperationen *und* Fehlerausgänge dargestellt. Die Beschreibung der Parameter einer Zugriffsoperation wird durch einen Verweis in die Liste der Parameterbeschreibungen realisiert. Die eigentliche Darstellung der D-Moduln durch Elemente der Liste DMOBESCHR beinhaltet zur Beschreibung der Zugriffsoperationen Verweise in die Liste BESCHROP. Die Beschreibung eines D-Moduls ist somit auf die drei Datenstrukturen DMOBESCHR, BESCHROP bzw. PARBESCHR verteilt.

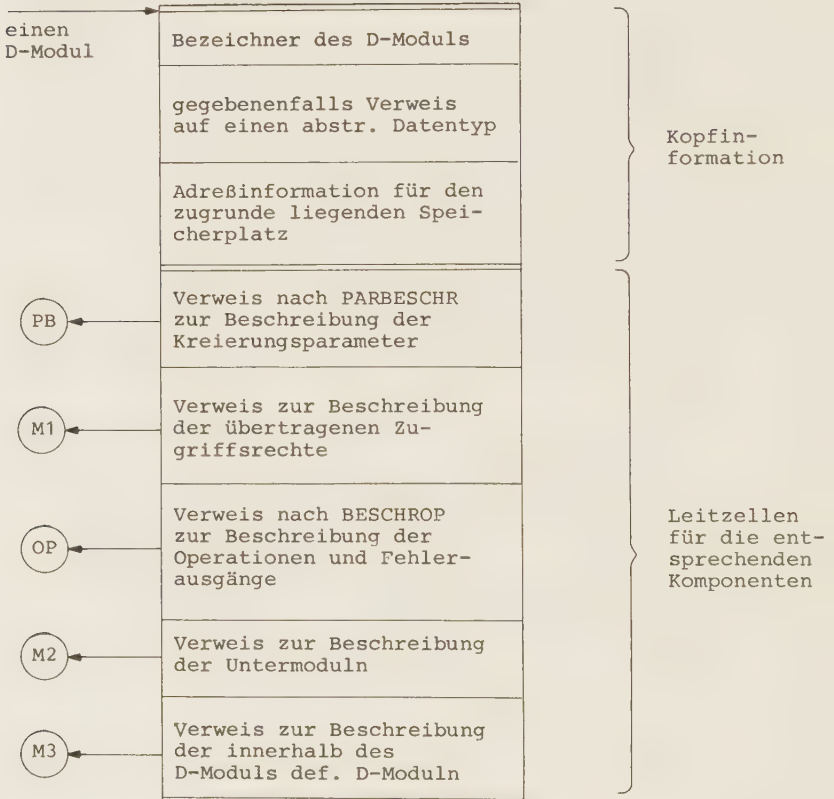
Eine gesonderte Führung der Listen BESCHROP und DMOBESCHR wurde aus folgenden Gründen gewählt:

- Die Vorgehensweise entspricht dem Grundkonzept der Trennung der Schnittstelle nach außen von der Internstruktur.
- Bei der Übertragung von Zugriffsrechten in der Schnittstelle von Moduln kann die Menge der Zugriffsoperationen, die auf den entsprechenden D-Moduln definiert sind, eingeschränkt werden. Die Beschreibung einer derartigen Schnittstelle muß daher auf einzelne Zugriffsoperationen verweisen können. Dies wird bei der Darstellung der Datenstruktur BESCHRÜZR noch deutlich werden.

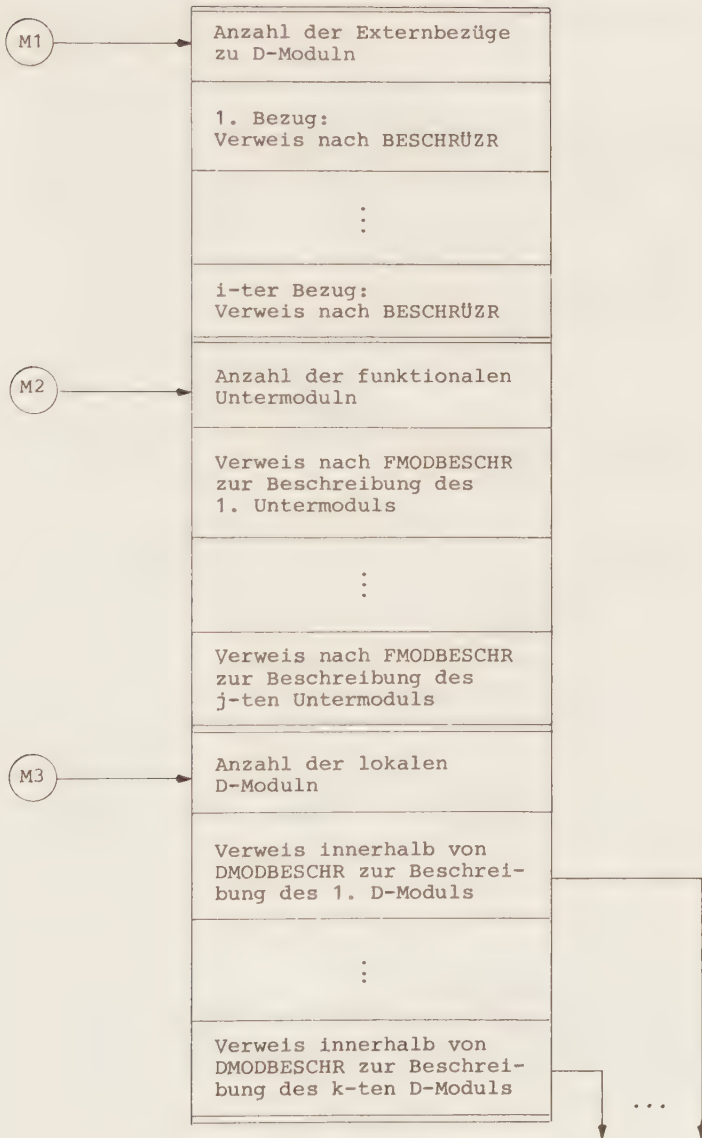
Es ergibt sich folgende Informationsdarstellung:

- Datenstruktur DMOBESCHR

Eintrag für

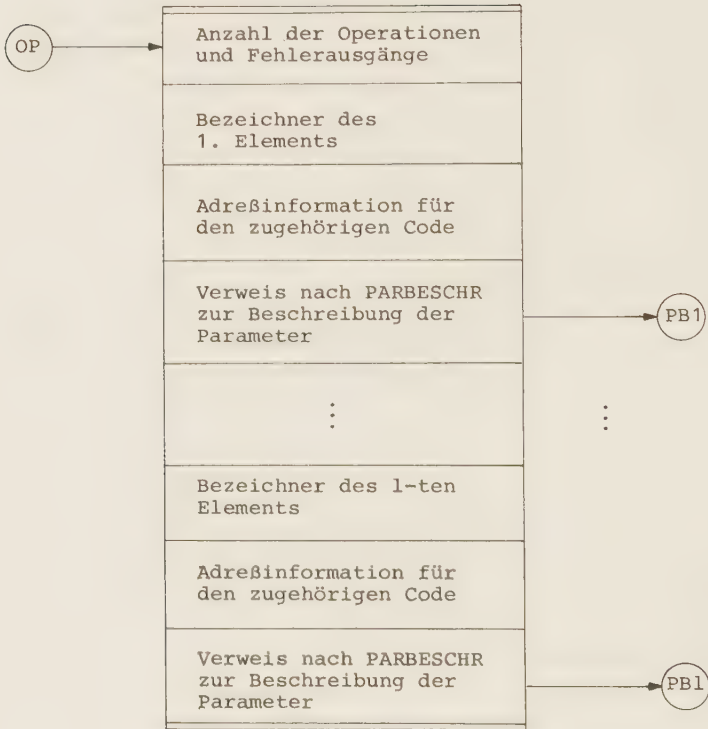


Figur 4/2a



Figur 4/2b

Datenstruktur BESCHROP



Figur 4/3

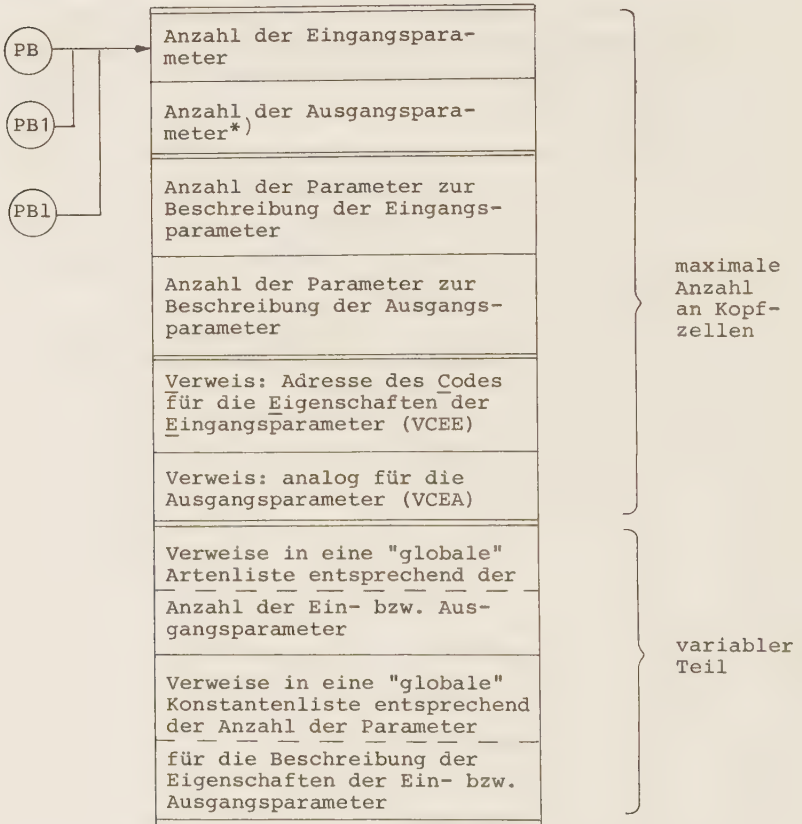
• Datenstruktur PARBESCHR

Bevor der Aufbau der Liste der Parameterbeschreibungen im Detail angegeben wird, sei nochmals an die Möglichkeiten der Parameterspezifikation im Rahmen der in Kapitel 3 definierten programmiersprachlichen Notation erinnert*):

- Die Spezifikation von Kreierungsparametern beinhaltet deren Art sowie die (eventuelle) Angabe von Eigenschaften der Parameter.
- Die E/A-Spezifikation von Zugriffsoperationen setzt sich wie folgt zusammen:
 - a) Angabe der (Ein- bzw. Ausgangs-)Parameter und deren Art
 - b) Angabe von Eigenschaften der Parameter
 - c) Angabe von Parametern für die Beschreibung der Eigenschaften.

*) Man vergleiche hierzu auch S. 95 (E/A-Spezifikation von Zugriffsoperationen) bzw. S. 82 (Beschreibung von Kreierungsparametern im Zusammenhang mit einer Datentyp-Implementierung).

Ein Element der Datenstruktur PARBESCHR kann folgendermaßen dargestellt werden:



Figur 4/4

*) Kreierungsparameter werden wie Eingangsparameter von Zugriffsoperationen behandelt. Beschreibt ein Element von PARBESCHR die Kreierungsparameter eines D-Moduls, so ist diese Angabe stets 0.

Es werden maximal 6, minimal 2 Kopfzellen für ein Element von PARBESCHR angelegt. Die Anzahl der tatsächlich eingerichteten hängt ab vom Inhalt der ersten beiden Zellen, die stets die Anzahl der Eingangs- (ANZE) bzw. Ausgangsparameter (ANZA) enthalten:

- Gilt ANZE=0 und ANZA=0, so werden 2 Kopfzellen angelegt.
 - Gilt ANZE=0 und ANZA≠0, so kann man mit 4 Zellen auskommen: Die dritte Zelle enthält die Anzahl der Parameter zur Beschreibung der Ausgangsparameter, die vierte den Verweis VCEA.
 - Analoges gilt für den Fall: ANZE≠0 und ANZA=0.
 - Gilt ANZE≠0 und ANZA≠0, so werden 6 Kopfzellen, wie in Figur 4/4 dargestellt, angelegt.
- Aufbau der Liste der übertragenen Zugriffsrechte
- Ein Element der Datenstruktur BESCHRÜZR entspricht der Beschreibung einer Parameterposition eines Moduls, auf der Zugriffsrechte bzgl. eines D-Moduls übertragen werden (*Zugriffsparameter*). Bei der E/A-Spezifikation der den Zugriffsrechten zugrunde liegenden Operationen müssen zwei Fälle unterschieden werden:

E/A-Spezifikation

- (A) bei der Übertragung eines Zugriffsrechtes
- (B) bei der Deklaration eines D-Moduls

Die Eigenschaften der Parameter im Fall (A) können gegenüber denen im Fall (B) eingeschränkt sein^{*)}.

^{*)} Man vergleiche hierzu im Beispiel 3.11 die Deklaration des D-Moduls ZENANDAT im F-Modul PRODSTEUER mit der Übertragung des Zugriffsrechtes (ZENANDAT, STORE AN) an den F-Modul VORVERARB.

Die Beschreibung der Elemente von BESCHRÜZR wird daher eine eigene, d.h. von speziellen D-Moduln unabhängige Parameterbeschreibung der entsprechenden Zugriffsoperationen enthalten.

Weiter ist zu beachten, daß ein formaler D-Modul-Name X in der "interface"-Komponente eines Moduls Y durchaus als aktueller Parameter bei der Spezifikation der Schnittstelle eines bzgl. Y untergeordneten Moduls Z verwendet werden kann.

Dieser Fall tritt ein, wenn ein von der Aufrufumgebung eines Moduls übertragenes Zugriffsrecht nach innen, d.h. an darunter liegenden Moduln, weitergereicht wird. In Beispiel 4.1 wird die soeben geschilderte Situation dargestellt.

Ein Element von BESCHRÜZR enthält folgende Information:

- Zusammenhang zwischen einem formalen und allen dynamisch möglichen aktuellen D-Modul-Namen bzgl. einer Parameterposition^{*)}
- Beschreibung der Zugriffsrechte

Aktuelle D-Modul-Namen werden realisiert durch Verweise in die Liste der D-Modul-Beschreibungen bzw. in die Liste BESCHRÜZR, wenn ein aktueller D-Modul-Name (wie in Beispiel 4.1, D-Modul X) ein formaler D-Modul-Name aus der "interface"-Komponente ist.

^{*)} Man vergleiche hierzu die Beispiele 3/8 und 3/9 auf den Seiten 93 bzw. 94.

Beispiel 4.1

```

function module Y(...,X,...);
  interface
    .
    .
    X with OP1,OP2;
    .

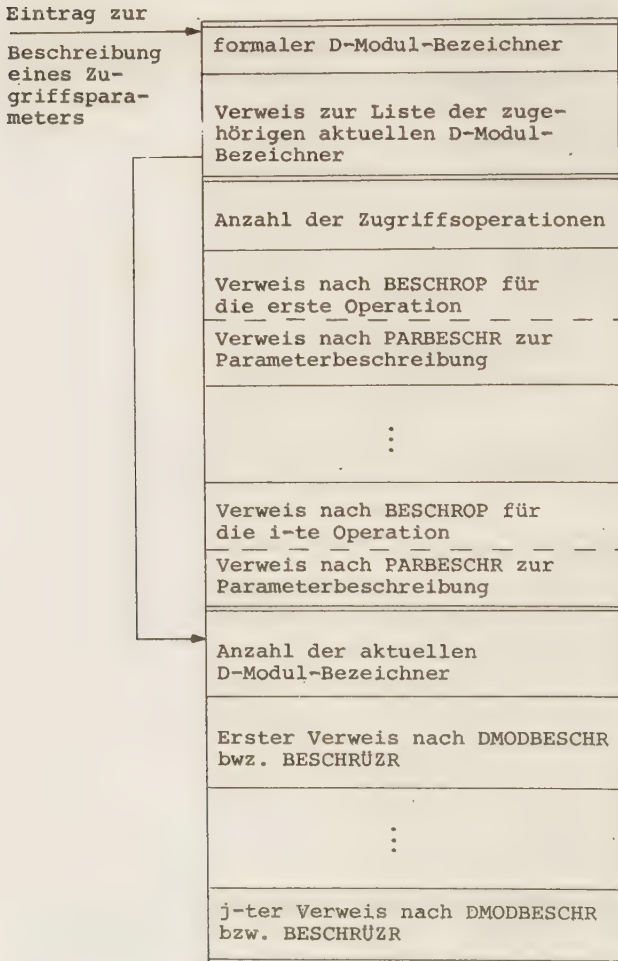
  end interface;
  .
  .
  submodules
    .
    .
    Z access to X with OP1;
    end access;
    .

  end submodules;
  .
  .
end function module Y;

```

□

Ein Element der Datenstruktur BESCHRÜZR kann nun wie folgt dargestellt werden:



Figur 4/5

Aus Tafel 4/1 ist ersichtlich, welche Eintragungen in die Liste der F-Modul-Beschreibungen der Modul F1 bewirkt. In den Tafeln wird generell für einen noch undefinierten Adreßbezug die Schreibweise undef verwendet. "Leere" Verweise werden durch nil gekennzeichnet.

Tafel 4/1

i=FMOBESCHR ₀	F1	}	Kopfin- formation
	<u>undef</u>		
	i+5	}	Leitzellen
	i+7		
	i+10		
i+5	1		
	BESCHRÜZR ₁		
i+7	2		
	FMOBESCHR ₁		
	FMOBESCHR ₂		
i+10	1		
	DMOBESCHR ₁		

□

Die zunächst noch offenen Verweise FMOBESCHR₁, FMOBESCHR₂ und DMOBESCHR₁ in Tafel 4/1 können erst nach der "Übersetzung" der F-Moduln F11($\hat{=}$ FMOBESCHR₁) und F12($\hat{=}$ FMOBESCHR₂) bzw. des D-Moduls D₂($\hat{=}$ DMOBESCHR₁) ergänzt werden.

Das Problem der offenen Verweise tritt auf, da das angewandte Auftreten von F- und D-Moduln (im Beispiel: F11, F12 bzw. D₂) in Untermodulspezifikationen bzw. D-Modul-Deklarationen (!) dem entsprechenden definierenden Auftreten in F- bzw. D-Modul-Implementierungen vorausgehen kann. Zur Behandlung des Problems bietet sich die aus der Übersetzung von (Vorwärts-)Sprüngen bekannte Vormerktechnik an^{*)}.

^{*)} vgl. etwa [SCH 75a].

Die Übersetzung der Schnittstellenbeschreibung von F1 führt zu einem Eintrag in die Liste der übertragenen Zugriffsrechte an der Stelle BESCHRÜZR₁.

In Tafel 4/2 ist die Eintragung für D1 in der Datenstruktur BESCHRÜZR dargestellt.

Tafel 4/2

j = BESCHRÜZR ₁	D1
	j+5
	2
	BESCHROP ₁
	PARBESCHR ₁
	BESCHROP ₂
	PARBESCHR ₂
j+5	1
	DMODBESCHR ₀

□

Der Verweis DMODBESCHR₀ in Tafel 4/2 bezieht sich auf den beim "Systemstart" dem (Haupt-)F-Modul F1 aktuell übergebenen D-Modul-Namen.

Die Verweise BESCHROP₁, PARBESCHR₁, PARBESCHR₂ in Tafel 4/2 werden hier nicht weiterverfolgt. Beispiele für Eintragungen in die Datenstrukturen BESCHROP bzw. PARBESCHR ergeben sich durch die nachfolgende, in Beispiel 4.3 angegebene "Verfeinerung" des Moduls F1 aus Beispiel 4.2.

Zur Demonstration des Zusammenspiels der Beschreibungslisten eines modularen Programmsystems wird im folgenden ein (abstraktes) Beispiel angegeben.

Beispiel 4.2

```

main function module F1(D1);

  interface
    D1 with OP11 (amode), OP12 returns amode;
  end interface;

  realization

  data
    provides data
      include D2(10)
        with OP21 (int I, bmode; 10 where  $1 \leq I \leq 10$ ),
          OP22 returns bmode;
    end data;

  submodules

    F11 access to
      D1 with OP11 (amode);
      D2 with OP22 returns bmode;
      end access;

    F12 access to
      D1 with OP11 (amode),
        OP12 returns amode;
      end access;

  end submodules;

  end realization;

end function module F1;

```

amode und bmode sthen stellvertretend für irgendeine Art.

□

Beispiel 4.3

- F-Modul-Implementierungen für F11 und F12

```

function module F11(DF1,DF2);

  interface
    DF1 with OP11 (amode);
    DF2 with OP22 returns bmode;
  end interface;

  :
end function module F11;

function module F12(DF3);

  interface
    DF3 with OP11 (amode),
        OP12 returns amode;
  end interface;

  :
end function module F12;

```

- D-Modul-Implementierung für D2

```

data module D2(int L);

  interface
    parameters . . . ;
    public procedures
      OP21 (int I, bmode; int K where  $1 \leq I \leq K$ ),
      OP22 returns bmode;
    end interface;

  realization
    :
  end realization;

end data module D2;

```

Die "realization"-Komponente des D-Moduls D2 ist gegeben durch *):

```

realization
  representation
    declare NICE_STACK(anystructure,L) KELLER
      with POP,TOP returns anystructure,
      PUSH(anystructure),
      EMPTY returns bool,
      errors OVERFLOW,UNDERFLOW;
    :
  end representation;
  submodules
    F2 access to
      KELLER with POP,EMPTY returns bool,
      errors UNDERFLOW
    end access;
  end submodules;
  public procedures
    OP21: ....;
    OP22: ....;
  end realization;

```

□

In den Tafeln 4/3 und 4/4 sind die Eintragungen dargestellt, die von der D-Modul-Implementierung für D2 aus Beispiel 4.3 bewirkt werden. Die Tafeln 4/5 und 4/6 beschreiben die Auswirkungen der F-Modul-Implementierungen für F11 und F12 auf die Datenstrukturen FMOBESCHR bzw. BESCHRÜZR.

*) Bei der Deklaration des D-Moduls KELLER wurde der in Beispiel 3.1 (Kap. 3.2, S. 84) definierte abstrakte Datentyp NICE_STACK verwendet.

Tafel 4/3

Eintragungen in DMOBESCHR

 $i = \text{DMOBESCHR}_1$

D2

nilundef PARBESCHR_3 nil BESCHROP_3 $i+8$ $i+10$ $i+8$

1

 FMOBESCHR_3 $i+10$

1

 DMOBESCHR_2

□

Tafel 4/4

Eintragungen in BESCHROP

i = BESCHROP₃ 2
 OP21
 undef
 PARBESCHR₄
 OP22
 undef
 PARBESCHR₅

Eintragungen in PARBESCHR

PARBESCHR ₃	1	PARBESCHR ₅	0
	0		1
	0		0
	<u>nil</u>		<u>nil</u>
	verw(<u>int</u>)		verw(<u>bmode</u>)
PARBESCHR ₄	2		
	0		
	1		
	<u>undef</u> (*)		
	verw(<u>int</u>)		
	verw(<u>bmode</u>)		
	verw(K)		

(*) Wird zu einem späteren Zeitpunkt, der Übersetzung durch einen Verweis auf ein Codestück für $1 \leq i \leq K$ ersetzt (vgl. Beispiel 4.3, data module D2)

Tafel 4/5

$i = \text{FMODBESCHR}_1$	F11
	<u>undef</u>
	$i+c_1$
	.
	.
	.
$i+c_1$	2
	BESCHRÜZR ₂
	BESCHRÜZR ₃
	.
	.
	.
	.
$j = \text{FMODBESCHR}_2$	F12
	<u>undef</u>
	$j+c_2$
	.
	.
	.
$j+c_2$	1
	BESCHRÜZR ₄

□

Tafel 4/6

$k = \text{BESCHRÜZR}_2$	DF1	$m = \text{BESCHRÜZR}_4$	DF3
	$k+5$		$m+7$
	1		2
	BESCHROP		BESCHROP
	OP11		OP11
	PARBESCHR ₆		PARBESCHR ₆
$k+5$	1		BESCHROP
	BESCHRÜZR ₁ $\otimes$		OP12
			PARBESCHR ₈
		$m+7$	1
$l = \text{BESCHRÜZR}_3$	DF2		BESCHRÜZR ₁
	$l+5$		
	1		
	BESCHROP ₃₊₄ $\oplus$		
	PARBESCHR ₇		
$l+5$	1		
	DMODBESCHR ₁ $\odot$		

$\otimes$ Verweis zu D1 in Tafel 4/1

$\oplus$ Verweis zu OP22 in Tafel 4/4

$\odot$ Verweis zu D2 in Tafel 4/3

□

Zur Bearbeitung der Datenstrukturen FMOBESCHR, DMOBESCHR, BESCHROP, PARBESCHR und BESCHRÜZR zur Übersetzungszeit könnte man nun D-Moduln definieren, deren Repräsentierungen aus eben diesen Datenstrukturen bestehen und mit entsprechenden Zugriffsoperationen versehen sind. Auf eine detaillierte Beschreibung derartiger Operationen soll aus Platzgründen verzichtet werden.

4.3 BESCHREIBUNG VON LAUFZEITASPEKTEN

Dieser Abschnitt befaßt sich mit der Speicherstruktur eines modularen Programmsystems zur Laufzeit.

Es werden dabei neben dem Aufbau von Versorgungsblöcken für F-Moduln bzw. für Zugriffsoperationen von D-Moduln insbesondere auch die Struktur von Speichersegmenten für die Implementierung der Repräsentierung von D-Moduln untersucht. In diesem Zusammenhang werden auch Überlegungen für die Realisierung von Zugriffsoperationen abstrakter Datentypen angestellt.

Bei der Organisation des Speichers zur Laufzeit unterscheiden wir folgende Segmente:

- Code für den einem F-Modul zugrunde liegenden Algorithmus
- Adreßbereich zur Darstellung eines D-Moduls
- Codestücke zur Realisierung von Zugriffsoperationen
- Versorgungsblöcke^{*)} zur Übertragung von Zugriffsrechten zwischen Moduln
- Bereich für Hilfszellen und Laufzeitkeller.

Betrachten wir zunächst die Versorgungsblöcke für Moduln (VBLM).

Diese Speicherbereiche enthalten zur Laufzeit die zur Ausübung der übertragenen Zugriffsrechte notwendige Information. Für jeden Zugriffsparameter steht eine Objektadresse (= Anfangsadresse eines Segmentes zur Darstellung eines D-Moduls), Verweise auf die Zugriffsoperationen und deren Parameterblöcke und die Adressen

^{*)} Auf die Beschreibung von Versorgungsblöcken für die als Unterprogramme realisierten Zugriffsoperationen sei hier verzichtet. Bekannte Techniken sind z.B. in [GÜS 72], Kap. 8 beschrieben.

der Fehlerausgänge zur Verfügung. Schließlich enthalten die Versorgungsböcke noch die Rückkehradresse zum aufrufenden Modul. Insgesamt ergibt sich der in Figur 4/6 dargestellte Aufbau.

Da diese Versorgungsböcke als "Bindeglieder" der Moduln untereinander dienen, müssen Adressen und Aufbau der Böcke den beiden jeweils beteiligten Moduln bekannt sein.

VBLM



Rückkehradresse
Anz. der Parameter
Adr(Obj. 1)
Anz. der Zugr.-Operationen
Adr(OP11)
Adr(Par.-Block 11)
⋮
Adr(OP1n)
Adr(Par.-Block 1n)
Anz. der Fehlerausgänge
Adr(Fehlerausg. 1)
⋮
Adr(Fehlerausg. k)
⋮
Adr(Obj. 1)
⋮

Figur 4/6

Zur Beschreibung von Adreßbereichen für D-Moduln bzw. des Codes für Zugriffsooperationen werden im folgenden Probleme bei der Implementierung abstrakter Datentypen untersucht.

Da der Code für die Zugriffsooperationen abstrakter Datentypen unter Umständen von mehreren Programmen verwendet wird, erscheint es sinnvoll, diese Operationen als eingriffsinvariante Unterprogramme (engl.: reentrant) zu realisieren. Hierzu wird der Programmbereich in einen variablen und einen konstanten Anteil zerlegt:

- Der variable Teil enthält
 - die Parameterversorgung,
 - Speicher für lokale Daten und Zwischenergebnisse und
 - Speicher für die Repräsentierung des jeweils bearbeiteten D-Moduls.
- Der konstante Teil enthält
 - eine Beschreibung des Aufbaus des Bereichs und
 - Befehlscode.

Der variable Adreßbereich einer Zugriffsooperation ist Teil des Speichers, der zur Darstellung eines entsprechenden D-Moduls gehört.

Die Gesamtdatenstruktur (GDS) eines D-Moduls setzt sich aus folgenden Komponenten zusammen:

- Rückkehradresse RADR
(nach Ausführung einer Zugriffsooperation)
- (- allgemeine Fehler-/Alarmadresse FADR)
- Parameterversorgungen für die Operationen
- Bereiche für lokale Daten der Operationen

- Zwischenergebniskeller
- Repräsentierung.

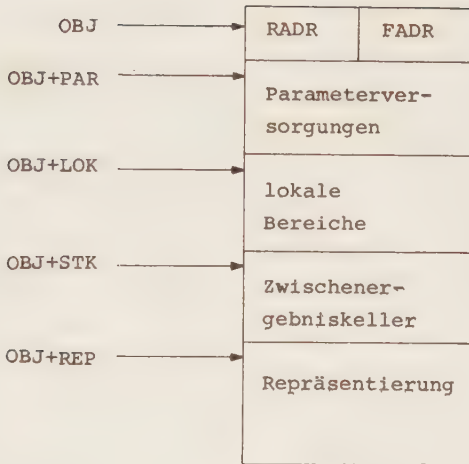
Da nie mit mehreren Operationen gleichzeitig auf einen D-Modul zugegriffen wird, genügt es, die Bereiche für Parameterversorgung, lokale Daten und Zwischenergebniskeller jeweils nur einmal (für alle Operationen gemeinsam) anzulegen. Es muß dann jeweils der maximal benötigte Platz eingerichtet werden.

Die Adressierung der Gesamtdatenstruktur von den Operationen aus geschieht über ein Basisregister (BASREG). Die einzelnen Komponenten der Gesamtdatenstruktur können über folgende konstante Distanzadressen angesprochen werden:

- PAR Anfang der Parameterblöcke (relativ zu
 .BASREG*)
- LOK Anfang der lokalen Bereiche
- STK Anfang des Zwischenergebniskellers
- REP Anfang der Repräsentierung.

In Figur 4/7 ist der schematische Aufbau eines (D-Modul-)Objektes mit der Anfangsadresse OBJ dargestellt.

*) Zur Unterscheidung zwischen Adresse (BASREG) und Inhalt (.BASREG) verwenden wir den "."-Operator aus BLISS (vgl. z.B. [WRH 71]).



Figur 4/7

Für die in den Beispielen 4.2 bzw. 4.3 beschriebene Modulstruktur ergibt sich die durch Figur 4/8 dargestellte Speicheraufteilung.

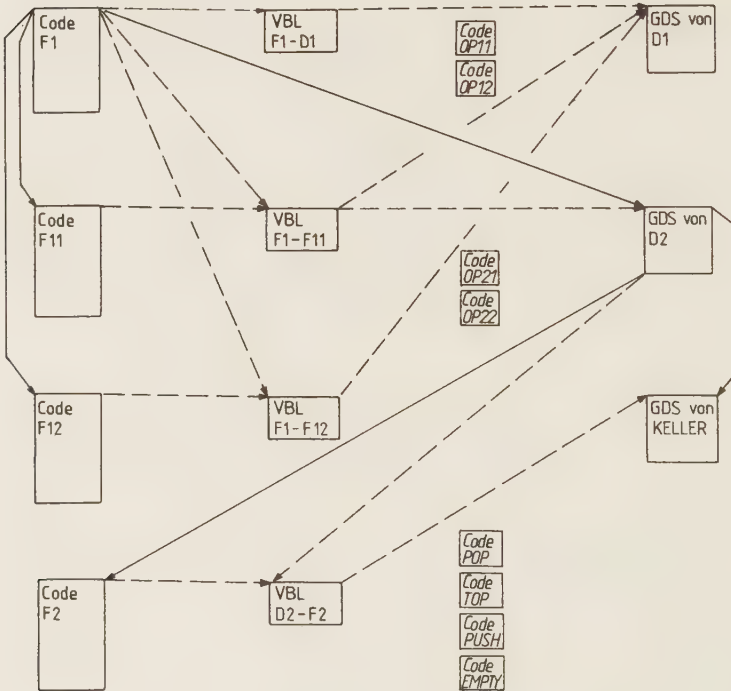


Fig. 4/8

Die Kanten zwischen den Speichersegmenten in Figur 4/8 stellen Adreßbezüge dar:

- Die durchgezogenen Kanten spiegeln die Systemhierarchie (vgl. rsys, Kap. 2.2, Def. 2.7) wider.
 - Die gestrichelten Kanten repräsentieren die in Figur 4/6 erläuterten Adreßbezüge.
 - Aus Gründen der Übersichtlichkeit wurden die Verweise von den Versorgungsblöcken zu den Codestücken der Operationen weggelassen.
- Zur Vervollständigung der Modulstruktur fehlen folgende Verbindungen:

von	nach
VBL F1-D1	OP11 u. OP12
VBL F1-F11	OP11 u. OP22
VBL F1-F12	OP11 u. OP12
VBL D2-F2	POP u. EMPTY.

Zum Schluß dieses Abschnitts sollen nun noch die Übersetzung eines F-Modul-Aufrufs und die Ausführung einer Zugriffsoperation angegeben werden.

- Übersetzung eines F-Modul-Aufrufs

call FM(OBJ1,..., OBJn)

Seien VBL_{FM} die Adresse des Versorgungsblockes für FM, OBJ1 bis OBJn die Objektanfangsadressen, dann ergibt sich folgende Übersetzung:

1. $VBL_{FM}+2 = OBJ1$
 $\vdots$
 $VBL_{FM}+c_1*n+c_2 = OBJn^*)$
 2. $VBL_{FM} = \text{Rückkehradresse RADR}$
 3. UP-Sprung nach FM
- Übersetzung der Ausführung einer Zugriffsoperation
 $OP(<\text{aktuelle Parameter}>) . OBJ$
1. Belege Basisregister BASREG mit OBJ
 2. Speichere Rückkehradresse RADR nach OBJ ($= .BASREG$)
 3. Belege Parameterversorgungsblock (beginnend bei $.BASREG+PAR$; vgl. Fig. 4/7) mit aktuellen Parametern
 4. UP-Sprung nach OP

Die folgenden Beispiele dienen zur Erläuterung des zur Bearbeitung der Datenmoduln verwendeten Adressierungsmechanismus.

- Auswahl des i-ten Parameters einer Zugriffsoperation
 $.BASREG+PAR+i$

*) c_1, c_2 sind zur Übersetzungszeit bekannte Konstanten und errechnen sich aus dem Aufbau des Versorgungsblockes; vgl. Fig. 4/6.

- Analoge Adreßrechnungen ergeben sich für Größen aus den Speicherbereichen, die durch LOK, STK bzw. REP gekennzeichnet sind:

.BASREG+LOK+j

.BASREG+STK+k

.BASREG+REP+l

Man erkennt, daß das Vorhandensein von indirekter (.BASREG) und modifizierter (+REP+l) Adressierung vorausgesetzt wird.

KAPITEL 5 MODULARE PROGRAMMIERUNG IN BLOCKORIENTIER- TEN SPRACHEN, FORTRAN UND ASSEMBLERN

5.1 ZIELSETZUNG, VORBEMERKUNGEN

In diesem Kapitel werden im wesentlichen zwei Ziele verfolgt.

Zum einen soll untersucht werden, welche Möglichkeiten die untersuchten Sprachen haben, um durch Einhalten einer entsprechenden Programmierdisziplin die Konstruktion von D- und F-Moduln nachzubilden. Aufgrund dieser Überlegungen soll zum anderen eine Wertung der Sprachen in bezug auf modulare Programmierung vorgenommen werden. Für den Entwurf großer Programmsysteme, und somit auch für die modulare Programmierung, spielt es u.a. eine Rolle, welche Möglichkeiten der Datenübertragung zwischen Programmeinheiten eine Sprache besitzt, wie man gemeinsam von mehreren Einheiten zu bearbeitende Daten organisiert bzw. inwieweit sich logisch verschiedene Systemteile textuell trennen und als voneinander unabhängig übersetzbare Programmteile darstellen lassen. Bevor nun für ALGOL 60^{*)}, FORTRAN und die Telefunken Assemblersprache TAS^{*)} Darstellungen von D- bzw. F-Moduln diskutiert werden, wollen wir für diese Sprachen die oben erwähnten Fragestellungen kurz behandeln. Hierzu ist in Tafel 5/1 eine Übersicht über die Möglichkeiten der Datenübertragung zwischen den in der jeweiligen Sprache definierten Programmeinheiten gegeben.

^{*)} ALGOL 60 steht stellvertretend für eine blockorientierte, TAS für eine Assemblersprache.

Tafel 5/1

Programmiersprache	Programmeinheiten	Datenübertragungsmöglichkeiten	Bemerkungen
ALGOL 60	Blöcke, Prozeduren	- Zwischen Blöcken gemäß der Blockstruktur - Zwischen getrennt übersetzbaren Prozeduren: nur über Parameter - Zwischen lokalen Prozeduren a) gemäß der Blockstruktur b) über Parameter	- Der "äußerste" Block entspricht dem Hauptprogramm - Die Lebensdauer von lokalen Variablen eines Blockes ist gleich der Ausführungsdauer dieses Blockes
FORTRAN	Hauptprogramm (HP) (SUBROUTINE-, FUNCTION-) Unterprogramme (UP) BLOCK DATA-Segmente	- Zwischen HP und UP sowie zwischen UP und UP: a) über Parameter b) über COMMON-Blöcke	- Die Variablen der COMMON-Blöcke werden statisch verwaltet
TAS	Teilprogramme, Segmente	- Zwischen Teilprogrammen: Explizite Kennzeichnung globaler Größen durch EINGG (definierend) bzw. EXTERN (angewandt) - Zwischen Segmenten eines Teilprogramms: Kennzeichnung von Segment-globalen Größen - Zwischen Segmenten <u>verschiedener</u> Teilprogramme können Daten nur über den EINGG-EXTERN-Mechanismus übertragen werden	- Mehrere Segmente werden jeweils zu einem übergeordneten Teilprogramm zusammengefaßt - Alle Teilprogramme werden getrennt übersetzt - Da Segmente und Teilprogramme auch als Unterprogramme ausgeführt sein können, ist eine Datenübertragung auch mit Hilfe von (Parameter-)Vorsorungsblöcken möglich

5.2 REALISIERUNG VON D-MODULN

In diesem Abschnitt sollen nun die Möglichkeiten untersucht werden, die sich in den drei zu untersuchenden Sprachen zur Darstellung *getrennt übersetzbarer* *) D-Moduln bieten.

Betrachten wir zunächst *ALGOL 60*.

Die Zugriffsoperationen eines D-Moduls werden als getrennt übersetzbare Prozeduren definiert. Da die Lebensdauer der einem D-Modul zugrunde liegenden Daten größer sein soll als die Ausführungszeit eines Prozedur- (bzw. Operations-)aufrufs müssen alle Daten der Repräsentierung im Hauptprogramm deklariert werden. Die Operationen bzw. die zugehörigen Prozeduren haben somit zwei Arten von Parametern:

- Alle Variablen der Repräsentierung, die von der jeweiligen Prozedur benötigt werden.
- Die "eigentlichen" Parameter der "public procedures".

Die Zugehörigkeit der Datendeklarationen bzw. der Prozeduren zu einem D-Modul muß durch Kommentar gekennzeichnet werden. Die Initialisierung der Repräsentierung kann direkt im Hauptprogramm erfolgen. Eine andere Möglichkeit besteht darin, eine "Initialisierungsprozedur" im Hauptprogramm aufzurufen.

*) Die Eigenschaft "getrennt übersetzbar" wurde bei dieser Untersuchung in den Vordergrund gestellt, da sie (zusammen mit der Forderung nach textueller Trennbarkeit von Programmeinheiten) beim Entwurf großer Programmsysteme eine sehr wesentliche Rolle spielt. Zur Erfüllung der übrigen Modul-Eigenschaften (vgl. Kap. 1.2) muß bei dieser Vorgehensweise verstärkt Programmierdisziplin geübt werden.

Beispiel 5.1

Im folgenden soll die Darstellung eines D-Moduls KELLER analog Beispiel 3.3 (Kap. 3, S. 86) in ALGOL 60 angegeben werden.

begin

comment HAUPTPROGRAMM;

comment DEKLARATION DER REPRAESENTIERUNG DES
D-MODULS KELLER.

ZUGRIFFSOPERATIONEN: PUSH, POP, TOP, EMPTY;

integer array STACK [1:100];

integer ST POINTER;

⋮

comment INITIALISIERUNG DES D-MODULS KELLER;

ST POINTER := 0;

⋮

end HAUPTPROGRAMM;

procedure POP(ST PTR);

comment "PUBLIC PROCEDURE" FUER DEN D-MODUL KELLER;

comment REPRAESENTIERUNG;

integer ST PTR;

begin

if ST PTR less 1

then ERROR("UNDERFLOW")

else ST PTR := ST PTR-1;

end POP;

```

procedure PUSH(ST,ST PTR,ELEM);
    comment "PUBLIC PROCEDURE" FUER DEN D-MODUL KELLER;
    comment REPRAESENTIERUNG;
    integer array ST;
    integer ST PTR;
    comment PARAMETER;
    integer ELEM;

    begin
        integer I;
        if ST PTR greater 100
            then ERROR("OVERFLOW")
            else
                begin comment KELLERN VON ELEM;
                    ST PTR := ST PTR+1;
                    ST[ST PTR] := ELEM;
                end;
            end PUSH;

```

Analog kann man Prozeduren zur Darstellung der Operationen TOP und EMPTY angeben.

Sei ACT ELEM von der Art integer.

Die Operation PUSH kann dann im Hauptprogramm wie folgt aufgerufen werden:

```
PUSH(STACK,ST POINTER, ACT ELEM) ;
```

□

Für die Programmiersprache *FORTTRAN* ergibt sich folgende Darstellung für D-Moduln:

Die Zugriffsoperationen werden als SUBROUTINE- bzw. FUNCTION-Unterprogramme definiert. Die Repräsentierung

wird in einem *benannten* COMMON-Block zusammengefaßt, den *alle* zu einem D-Modul gehörenden Unterprogramme (Operationen) beinhalten.

Legt die Realisierung einer Zugriffsoperation die Definition weiterer Unterprogramme ("local procedures", vgl. Kap. 3.2) nahe, so muß unter Umständen auch in diesen der entsprechende COMMON-Block vereinbart werden. Zum Datenaustausch zwischen "local" und "public procedures" empfiehlt sich häufig auch die Einrichtung "lokaler" COMMON-Bereiche.

Die Parameter der Zugriffsoperationen werden durch Parameter der entsprechenden Unterprogramme dargestellt. Werden bei der Initialisierung der Repräsentierung nur Konstanten verwendet (statische Vorbesetzung), so kann dies in einem BLOCK DATA-Segment erfolgen. Bei Initialisierung mit variablen Größen (dynamische Vorbesetzung) muß ein spezielles Initialisierungs-Unterprogramm angelegt werden.

Beispiel 5.2

Analog Beispiel 5.1 soll die Realisierung des D-Moduls KELLER in FORTRAN angegeben werden^{*)}.

```
C      HAUPTPROGRAMM
C      :
C      :
C      ES WIRD EIN D-MODUL MIT DEM NAMEN KELLER MIT
C      DEN ZUGRIFFSOPERATIONEN PUSH, POP, TOP, EMPTY
C      VERWENDET
C      :
C      :
C      END
```

^{*)} Bei der Realisierung der Algorithmen wurden Anweisungsformen eines erweiterten FORTRAN, wie in [MEI 77] angegeben, verwendet.

```

BLOCK DATA

C  INITIALISIERUNGSSEGMENT

INTEGER STACK,STPTR,MAXSTK
DIMENSION STACK(100)
COMMON /KELLER/STACK,STPTR,MAXSTK
DATA STPTR,MAXSTK/0,100/

END

SUBROUTINE POP

C  ZUGRIFFSOPERATION FUER DEN
C  D-MODUL KELLER

COMMON /KELLER/STACK,STPTR,MAXSTK

IF (STPTR .LT. 1) THEN
    CALL ERROR("UNDERFLOW")
ELSE
    STPTR = STPTR-1
END IF
RETURN

END

SUBROUTINE PUSH(ELEM)

C  ZUGRIFFSOPERATION FUER DEN
C  D-MODUL KELLER

INTEGER ELEM
COMMON /KELLER/STACK,STPTR,MAXSTK

IF (STPTR .GE. MAXSTK) THEN
    CALL ERROR("OVERFLOW")
ELSE
    STPTR = STPTR+1
    STACK(STPTR) = ELEM
END IF
RETURN

END

```

Analog lassen sich Darstellungen für die Operationen TOP und EMPTY angeben. Sei ACTEL als "INTEGER" deklariert, dann kann die Operation PUSH im Hauptprogramm wie folgt aufgerufen werden:

CALL PUSH(ACTEL)

Für die Realisierung von D-Moduln in *TAS* ergeben sich prinzipiell zwei Möglichkeiten:

- Der gesamte D-Modul wird als Teilprogramm realisiert. Dieses beinhaltet zur Darstellung der Repräsentierung die Belegung der entsprechenden Arbeitsspeicherplätze. Neben einem Initialisierungssegment umfaßt das Teilprogramm außerdem für jede Zugriffsoperation ein weiteres benanntes Segment. Diese werden vom Hauptprogramm als Unterprogramme aufgerufen.

Zur Bearbeitung müssen die einzelnen Komponenten der Repräsentierung in allen Segmenten global verfügbar sein.

- Die Definition und Initialisierung der Repräsentierung bilden ein Teilprogramm. Die einzelnen Komponenten der Datenstruktur werden durch den Pseudobefehl EINGG als globale Größen gekennzeichnet.

Die einzelnen Zugriffsoperationen werden ebenfalls als Teilprogramme realisiert und vom Hauptprogramm als Unterprogramme aufgerufen. Die von der jeweiligen Zugriffsoperation benutzten Teile der Repräsentierung werden in den entsprechenden Teilprogrammen durch den Pseudobefehl EXTERN verfügbar gemacht.

Welche der beiden geschilderten Möglichkeiten im Einzelfall zu wählen ist, hängt vom Umfang des D-Moduls ab. Handelt es sich um komplexe Zugriffsoperationen, so ist im allgemeinen die zweite Alternative vorzuziehen. In den Beispielen 5.3 bzw. 5.4 werden analog zu Beispiel

5.1 zwei alternative Darstellungen des D-Moduls KELLER in TAS angegeben. In beiden Fällen müssen die Namen der Unterprogramme (Operationen) durch den Pseudobefehl EXTERN in den aufrufenden Programmeinheiten bekannt gemacht werden.

Welcher Parametermechanismus bzw. welcher Aufbau der Parameter-Schlüsselinformation für die Unterprogramme gewählt wird, hängt von den speziellen Gegebenheiten ab. Eine Übersicht über bekannte Techniken findet man in [GÜS 72], Kap. 8.

Beispiel 5.3

```

KELLER = SEGM,
    -- Teilprogramm zur Darstellung eines
    D-Moduls KELLER --
    EINGG(INITST,PUSH,POP,TOP,EMPTY),
    -- Belegung vom Arbeitsspeicher --
    STACK .= ASP 100,
    STPTR .= ASP 1,
    MAXSTK .= ASP 1,

INITST = SEGM,
    -- Unterprogramm zur Initialisierung der
    Repräsentierung --
    .
    .
    .
    -- UP-Rücksprung --
    -- END INITST --

PUSH = SEGM,
    -- Zugriffsoperation für den D-Modul KELLER --
    .
    .
    .
    -- UP-Rücksprung --
    -- END PUSH --

POP = SEGM,
    .
TOP = SEGM,
    .
EMPTY = SEGM,
    .
    ENDE,

```

□

Die Verwendung des D-Moduls KELLER gemäß Beispiel 5.3 in einem TAS-Programm PROG ist dann wie folgt:

```

PROG = SEGM,

-- Es wird ein D-Modul KELLER mit den Zugriffs-
   operationen (INITST), PUSH, POP, TOP, EMPTY
   verwendet --

EXTERN KELLER(INITST,PUSH,POP,TOP,EMPTY),

    .
    .
    .
    .

-- Parameterversorgung für PUSH --

    .
    .
    .

SU PUSH,

    .
    .
    .

ENDE,

```

□

Beispiel 5.4

```

KELLER = SEGM,
    -- Teilprogramm für die Definition der Reprä-
    -- sentierung und die Initialisierung --
    EINGG(INITST,STACK,STPTR)
    -- Belegung von Arbeitsspeicher --
    STACK .= ASP 100,
    STPTR .= ASP 1,
    MAXST .= ASP 1,
INITST = SEGM,
    -- Unterprogramm zur Initialisierung der
    -- Repräsentierung
    -- UP-Rücksprung --
    -- END INITST --
    -- END KELLER --
ENDE,

PUSH = SEGM,
    -- Teil- bzw. Unterprogramm zur Realisierung
    -- der Zugriffsoperation PUSH des D-Moduls
    -- KELLER --
    EINGG(PUSH),
    EXTERN KELLER(STACK,STPTR),
    -- PUSH verändert sowohl die Variable STACK
    -- als auch STPTR --
    -- Algorithmus für PUSH --
    -- UP-Rücksprung --
    -- END PUSH --
ENDE,

```

```

POP = SEGM,
    -- Teil- bzw. Unterprogramm zur Realisierung der
        Zugriffsoperation POP des D-Moduls KELLER --
    EINGG(POP),
    EXTERN KELLER(STPTR),
    -- POP verändert nur den Kellerzeiger STPTR --
    -- Algorithmus für POP --
    -- UP-Rücksprung --
    -- END POP --
    ENDE,

```

Analoge Darstellungen ergeben sich für die Zugriffsoperationen TOP und EMPTY. □

Die Verwendung des D-Moduls KELLER gemäß Beispiel 5.4 in einem TAS-Programm PROG ist dann wie folgt:

```

PROG = SEGM
    -- Es wird ein D-Modul KELLER mit den Zugriffs-
        operationen (INITST), PUSH, POP, TOP, EMPTY
        verwendet --
    EXTERN KELLER(INITST),
    EXTERN PUSH,
    EXTERN POP,
    EXTERN TOP,
    EXTERN EMPTY,
    -- Parameterversorgung für PUSH --
        :
    SU PUSH,
        :
    ENDE,

```

□

5.3 REALISIERUNG VON F-MODULN

In den drei zu untersuchenden Sprachen ALGOL 60, FORTRAN und TAS bieten sich sehr unterschiedliche Möglichkeiten zur Realisierung der in den Kapiteln 1.3 bzw. 3.3 beschriebenen F-Moduln.

Generell müssen, wie auch bei der Darstellung von D-Moduln, fehlende programmiersprachliche Konstruktionen durch entsprechende Programmierdisziplin ersetzt werden. Ziel dieses Abschnitts ist die Angabe von Konstruktionen zur Programmierung getrennt übersetzbarer F-Moduln in den drei oben erwähnten Sprachen.

- In ALGOL 60 werden F-Moduln durch getrennt übersetzbare Prozeduren dargestellt. Die zur Bearbeitung *übertragenen Daten* werden als Parameter übergeben. Die entsprechenden Zugriffsoperationen können ebenfalls Parameter sein. Die formale Korrektheit dieser Prozedurparameter beim Aufruf eines F-Moduls kann zur Übersetzungszeit jedoch nicht geprüft werden, da ALGOL 60 keine vollständige Parameterspezifikation verlangt. Eine andere Möglichkeit ergibt sich durch die Deklaration der Zugriffsoperationen als Code-Prozeduren im "Rumpf des F-Moduls" bzw. der den F-Modul realisierenden Prozedur. In beiden Fällen ist es ratsam, die Zusammengehörigkeit der Daten und der entsprechenden Prozeduren (Zugriffsoperationen) durch Kommentar zu kennzeichnen.

Die *Repräsentierungen lokaler D-Moduln* werden im "F-Modul-Rumpf" deklariert. Die Realisierung dieser D-Moduln erfolgt im übrigen gemäß Kap. 5.2. Untergeordnete F-Moduln werden als Codeprozeduren ausgeführt, die an diese Moduln übertragenen Zugriffsrechte durch Kommentar beschrieben.

- Die Darstellung von F-Moduln in *FORTTRAN* durch SUBROUTINE-Unterprogramme unterscheidet sich von der in ALGOL 60 im wesentlichen nur durch die günstigere Realisierung lokaler D-Moduln: Die Repräsentierungen dieser D-Moduln brauchen *nicht* im aufrufenden F-Modul deklariert zu werden (vgl. Kap. 5.2).
Da die Realisierung der Repräsentierungen von D-Moduln durch benannte COMMON-Blöcke erfolgt und somit die dort gehaltenen Daten statisch verwaltet werden, ist bei lokalen D-Moduln darauf zu achten, daß die Gültigkeit der Repräsentierungen gleich der Ausführungsdauer des F-Moduls ist. Es muß also dafür gesorgt werden, daß bei jedem F-Modul-Aufruf alle lokalen D-Moduln von neuem initialisiert werden.

- In *TAS* ist eine Realisierung von F-Moduln durch Teilprogramme naheliegend.
Neben der Kennzeichnung der Schnittstellen nach außen und innen durch Kommentar müssen die daraus resultierenden (Adreß-)Bezüge in *TAS* durch den Pseudobefehl EXTERN spezifiziert werden.
Entsprechend der in Kap. 4.3 beschriebenen Implementierung von F-Moduln bzw. deren Aufruf werden die Adressen der den Zugriffsrechten zugrunde liegenden D-Moduln bzw. Operationen als Parameter übergeben (vgl. Aufbau eines Versorgungsblockes gemäß Fig. 4/6, Kap. 4.3, S. 132).

Zur Verdeutlichung sollen im folgenden Darstellungen des F-Moduls F1 aus Beispiel 4.2, Kap. 4.2, S. 123 in ALGOL 60, *FORTTRAN* und *TAS* angegeben werden.

- ALGOL 60

```

procedure F1(D1);
    comment Spezifikation von D1;
        . . .
    begin
        comment "Schnittstelle nach außen";
            comment Zugriff zu einem D-Modul D1 mit der
                Operation OP11;
            procedure OP11; code;
            comment Parameterbeschreibung für OP11;
                . . .
        comment "Schnittstelle nach innen";
        comment "provided data";
            comment - Deklaration der Repräsentierung
                des D-Moduls D2
                . . .
            - Deklaration der zugehörigen
                Operationen;
                procedure OP21; code;
                procedure OP22; code;
        comment "submodules";
            procedure F11; code;
                comment Schnittstellenbeschreibung für F11;
                    . . .
            procedure F12; code;
                comment Schnittstellenbeschreibung für F12;
                    . . .
        comment "algorithm";
            F11(...);
            F12(...);
    end F1;

```

Beim Aufruf des F-Moduls F11 sind die Repräsentierungen von D1 und D2, beim Aufruf von F12 die von D1 als Parameter anzugeben.

- FORTRAN

```

      SUBROUTINE F1
C      SCHNITTSTELLE NACH AUSSEN
C      ZUGRIFF ZU EINEM D-MODUL D1 MIT DER OPERATION
C      OP11.
      <PARAMETERBESCHREIBUNG VON OP11>
C      SCHNITTSTELLE NACH INNEN
C      "PROVIDED DATA"
C      DEFINITION EINES D-MODULS D2 MIT DEN
C      ZUGRIFFSOPERATIONEN OP21, OP22
C      <PARAMETERBESCHREIBUNG FUER OP21, OP22>
C      "SUBMODULES"
C      VERWENDUNG VON F-MODULN F11, F12
C      <SCHNITTSTELLENBESCHREIBUNG FUER F11 UND F12>
C      INITIALISIERUNG VON D2
      CALL INITD2(...)
C      "ALGORITHM"
      :
      :
      CALL F11
      :
      :
      CALL F12
      :
      :
      RETURN
      END

```

Beim Aufruf der F-Moduln F11 bzw. F12 sind i.a. keine Parameter anzugeben. Werden (logisch gesehen) auf *einer* F-Modul-Parameterposition Zugriffsrechte bzgl. *mehrerer verschiedener, gleichartiger* D-Moduln übertragen (vgl. Beispiel 3.8, Kap. 3.3, S. 93), so muß dies durch entsprechende Steuerparameter realisiert werden.

- TAS

```

F1 = SEGM,

    EINGG(F1),

    -- Schnittstelle nach außen --

    -- Zugriff zu einem D-Modul D1 mit der
      Operation OP11 --

    -- Übertragen der Schlüsselinformation
      bezüglich des Zugriffsrechtes (D1,OP11)
      in einen lokalen Kontrollblock*) --

    -- Schnittstelle nach innen --

    -- "provided data" --

    -- Definition eines D-Moduls D2 mit den
      Operationen OP21,OP22 --

    EXTERN D2(INITD2,OP21,OP22)**),

    -- Initialisierung von D2 --

    -- Unterprogrammsprung INITD2 --

```

*) Diese Vorgehensweise entspricht einer in Assemblersprachen üblichen Unterprogrammtechnik.

**) Wir gehen hier davon aus, daß der gesamte D-Modul D2 durch ein TAS-Teilprogramm realisiert ist (vgl. Beispiel 5.3, Kap. 5.2, S. 149).

```

-- "submodules" --

-- Verwendung von F-Moduln F11,F12 --

-- Schnittstellenbeschreibungen für F11 und F12 --

EXTERN F11,
EXTERN F12,

-- "algorithm" --
.
.
.
-- Aufruf von F11*) --

    -- 1. Lade Adressen von D1,D2,OP12,OP22
        in den Versorgungsblock von F11*) --

    2. Unterprogrammsprung F11 --
    .
    .
    .

-- END F1 --

ENDE,

```

Zusammen mit den Überlegungen in Kap. 5.2 kann nun folgendes festgestellt werden:

Durch den sehr schwerfälligen Datenübertragungsmechanismus und die blockstrukturabhängige Lebensdauer von Daten in ALGOL 60 wird diese Sprache zum Entwurf modular aufgebauter Software-Produkte unbrauchbar.

FORTRAN hingegen kommt, bedingt durch das COMMON-Konzept, im allgemeinen den Anforderungen der modularen Program-

*) Man vergleiche hierzu die Übersetzung eines F-Modul-Aufrufs in Kap. 4.3, S. 137.

mierung besser entgegen.

In Assembler-Sprachen kann zwar das vorgestellte Modul-Konzept am genauesten nachempfunden werden, fehlende Kontrollstrukturen und das niedrige Programmierniveau schmälern jedoch diesen Vorteil.

KAPITEL 6 GRENZEN DES KONZEPTES

Das nachfolgende Kapitel behandelt Probleme, die sich im Zusammenhang mit mehrstelligen Operationen auf D-Moduln bzw. mit dem textuellen Auslagern der Implementierungen von F- und D-Moduln (vgl. Kap. 3.2) ergeben. Der Begriffsbildung in [JÄP 77] folgend, wird anhand des "prison mail system"-Problems aus [AMH 77] untersucht, welche Möglichkeiten zur Realisierung von Schutzsystemen das vorgestellte Modul-Konzept bietet.

Betrachten wir zunächst das Problem der *mehr- bzw. zweistelligen Operationen* auf D-Moduln.

Seien durch $DM_1 = (id_1, Zop_1, Rep_1)^*)$ bzw.

$DM_2 = (id_2, Zop_2, Rep_2)$ zwei D-Moduln gegeben. Gesucht ist die Implementierung einer zweistelligen Operation**)

$z: DM_1 \times DM_2 \rightarrow DM_1 \times DM_2$. Hierzu bieten sich verschiedene Lösungsvorschläge an.

- Das vorgestellte Modul-Konzept erlaubt zunächst nur eine Realisierung von z durch ausschließliche Verwendung der Zugriffsoptionen aus Zop_1 bzw. Zop_2 . Diese "saubere" Lösung kann sehr ineffizient sein.
- Implementiert man z als Zusatzoperation bzgl. einer der beiden D-Moduln, d.h.

$$z: Rep_1 \times DM_2 \rightarrow Rep_1 \times DM_2,$$

so erhält man eine "saubere" Lösung, die aber nur effizient bzgl. eines Moduls (hier DM_1) ist.

*) (id, Zop) sei die von einem D-Modul zur Verfügung gestellte Schnittstelle nach außen (vgl. auch Kap. 2.2, S. 51),
Rep die zugrunde liegende Repräsentierung.

**) Eine zweistellige Operation z kann z.B. ein Test sein zur Bestimmung der Gleichheit zweier Keller (vgl. Beispiel 3.3, Kap. 3.2).

- Eine weitere Möglichkeit besteht darin, daß z durch Programmierung auf "formalen Repräsentierungen" Rep_1 , Rep_2 , d.h. getrennt von den D-Moduln DM_1 , DM_2 realisiert wird.

Dies kann man sich folgendermaßen vorstellen:

z entspricht einer Prozedur oder Funktion, wobei die Arten der (zwei) formalen Parameter mit den "Arten der Repräsentierungen" von DM_1 bzw. DM_2 übereinstimmen.

Diese wohl effiziente Lösung hat den Nachteil, daß hierdurch die Bearbeitung einer Repräsentierung *unabhängig* von dem zugehörigen D-Modul gestattet wird.

- Die Operation z kann auch implementiert werden, indem man DM_1 und DM_2 zu *einem* D-Modul DM , durch "Vereinigung" von Rep_1 und Rep_2 , zusammenfaßt und somit z eine Zugriffsoperation bzgl. DM darstellt.

Es gibt einige Gründe, die gegen diese Vorgehensweise sprechen:

1. Die "Vereinigung" von D-Moduln führt unter Umständen zu sehr umfangreichen Programmbausteinen.
2. Komplexe D-Moduln widersprechen dem Konzept der kontrollierten Informationsverteilung (vgl. [PAR 71]), wenn die einzelnen Repräsentierungen der "ursprünglichen D-Moduln" keine logischen Beziehungen untereinander haben, sondern nur aus Effizienzgründen als eine Einheit betrachtet werden.

In konkreten Anwendungen wird man häufig eine Lösung wählen, die einer der beiden zuerst genannten Möglichkeiten entspricht^{*)}. Während der vierte Vorschlag mit Einschränkungen noch akzeptiert werden kann, ist die

^{*)} Es sei vermerkt, daß das Konzept der F-Moduln eben gerade zur Implementierung mehrstelliger oder übergeordneter Funktionen bzgl. D-Moduln unter Verwendung der entsprechenden D-Modul-Operationen dient (vgl. Lösung 1).

dritte Alternative abzulehnen.

Die Implementierung der F- bzw. D-Moduln eines Systems durch (textuell) getrennte Programmsegmente (vgl. Kap. 3.2, S. 86 ff) kommt dem Ziel der modularen Programmierung "Zerlegung in überschaubare, handhabbare Einheiten" sehr entgegen. Zum anderen bringt diese Vorgehensweise aber auch Probleme mit sich:

Das textuelle Auslagern der Implementierung eines Moduls M aus einem entsprechend übergeordneten M' hat zur Folge, daß M auch "außerhalb" von M' sichtbar wird. Man möchte jedoch vermeiden, daß M dadurch automatisch in jedem anderen Modul verwendbar wird.

Um die Eindeutigkeit der Modul-Namen in einem System zu gewährleisten, sind deshalb bei der Namensgebung für die ausgelagerten Moduln Absprachen zu treffen, die im Widerspruch zur Kontextunabhängigkeit stehen.

Das Problem der Namenskollision ließe sich im Zusammenhang mit D-Moduln wie folgt lösen:

Alle ausgelagerten D-Moduln werden mit Hilfe von abstrakten Datentypen realisiert (vgl. D-Modul-Deklaration, Kap. 3.2, S. 87).

Damit sind die Namen der auf diese Weise deklarierten D-Moduln nur noch lokal in *einem* Modul sichtbar.

Als nachteilig muß empfunden werden, daß bei dieser Lösung zur Definition der System-D-Moduln jeweils ein abstrakter Datentyp (durch einen type module, vgl. Kap. 3.2, S. 82) implementiert werden muß, obwohl dieser Typ jeweils nur einmal in einer entsprechenden D-Modul-Deklaration verwendet wird.

Eine analoge Lösung des Namen-Problems für F-Moduln würde ein Konzept zur Implementierung "abstrakter Funktionstypen" (ähnlich der Prozedur-Typen in ALGOL 68) voraussetzen. Die Spezifikation untergeordneter F-Moduln dürfte in diesem Fall dann ausschließlich mit Hilfe derartiger Funktionstypen erfolgen. Diese Lösung ist jedoch aus praktischen Erwägungen zu schwerfällig. Der Nachteil der Namenskollisionen ist demgegenüber vorzuziehen.

In *Schutzsystemen* betrachtet man handelnde Komponenten, sogenannte *Subjekte*, und die zu verarbeitenden Daten, sogenannte *Objekte*. Auf den Objekten, die gewisse *Informationen* enthalten, sind *Zugriffsrechte* definiert, die von den Subjekten ausgeübt werden dürfen. Ein *Regelsystem* legt nun fest, in welcher Weise die Subjekte auf die Objekte zugreifen dürfen, bzw. welche Informationen für welches Subjekt bestimmt sind.

Will man nur die Verteilung der Zugriffsberechtigung auf die Objekte in bezug auf die Subjekte regeln, so spricht man von einem *Objektschutzsystem*. Der Inhalt der Objekte spielt bei dieser Betrachtungsweise keine Rolle. Will man dagegen auch auf den Informationsfluß einwirken, so spricht man von einem *Informationsschutzsystem*.

Der Unterschied wird am deutlichsten, wenn man sich die Arbeitsweise der beiden Schutzmechanismen ansieht (vgl. [JÄP 77]). Im ersten Fall heißt das Arbeitsprinzip: "Kein Subjekt darf ohne Erlaubnis auf ein Objekt zugreifen". Im zweiten Fall heißt es: "Kein Subjekt darf Information erhalten, die nicht für dieses Subjekt bestimmt ist".

In dem von uns vorgestellten Graphmodell eines modularen Programmsystems entsprechen die F-Moduln den Subjekten und die D-Moduln den Objekten. Das Regelsystem ist gegeben durch die in den Definitionen 2.1 bis 2.11 aus

Kapitel 2 festgelegten Eigenschaften und Einschränkungen der Kanten zwischen F- und D-Moduln.

Anhand der nachfolgenden Problemstellung (vgl. [AMH 77]) soll untersucht werden, welche Möglichkeiten zur Realisierung von Schutzsystemen das vorgestellte Modul-Konzept bietet.

Beschreibung eines Gefängnispostsystems:

1. Die Gefangenen (engl.: prisoner) werden in Einzelhaft gehalten und sind auf diese Weise nicht in der Lage, direkt oder indirekt miteinander zu kommunizieren, außer durch das Gefängnispostsystem.
2. Die Gefangenen tauschen mittels des Postsystems Botschaften aus, die von den Gefängniswärtern (engl.: guard) abgeholt und zugestellt werden.
3. Jede Botschaft besteht aus einer Mitteilung und einer Unterschrift und wird in einen adressierten und verklebten Umschlag gelegt.

Bis auf die Tatsache, daß sich Postkunde (Gefangener) und Briefträger (Gefängniswärter) "gegenseitig verdächtigen", ähnelt dieses System dem normalen Postsystem. Zum Schutz der Botschaften gibt es eine gegenseitige Übereinkunft über eine Sicherheitsbedingung, die stets beachtet wird:

Nur der Adressat darf einen verklebten Umschlag öffnen.

Um nun zu verhindern, daß die Gefängniswärter aus den Botschaftenbewegungen zuviel Information ableiten können, haben die Gefangenen folgende Konvention ausgearbeitet:

4. Jeder Gefangene faßt die während eines gewissen Zeitraumes zu versendenden Botschaften in einem weiteren verklebten Umschlag zu einem (möglicherweise leeren) Bündel (engl.: bundle) zusammen und adressiert dieses

an einen unparteiischen Postmeister (engl.: postmaster).

5. Der Postmeister öffnet die Bündel, sortiert die Briefe nach entsprechenden Gefangenenadressen, faßt diese zu einem neuen (evtl. leeren) verklebten Bündel, mit der Adresse des Gefangenen, zusammen und übergibt sie dem Gefängniswärter zur Zustellung.

Während eines Zustellungszyklus sendet und empfängt auf diese Weise jeder Gefangene genau ein Bündel.

In Figur 6/1 ist dargestellt, welche Situation sich zur Verteilung der Briefbündel ergibt. Jedem Gefangenen ist ein Briefkasten (engl.: mailbox), dem Postmeister ein Hauptbriefkasten (im Bild "MAINBOX" genannt) zugeordnet.

Die Gefangenen, der Wächter und der Postmeister sind als F-, die Briefkästen als D-Moduln dargestellt.

Durch die Schreibweise "bundle1" bzw. "bundle2" soll der Unterschied der Briefbündelarten angezeigt werden.

bundle1 bedeutet: adressiert an den Postmeister

bundle2 bedeutet: adressiert an einen Gefangenen.

Faßt man nun die Objekte der Art bundle1 bzw. bundle2 als D-Moduln auf, so sind mit diesen verschiedene Zugriffsrechte verbunden:

Während der Wächter (GUARD) die ihm zur Verfügung gestellten Briefbündel^{*)} nur weitertragen darf, haben der Postmeister und die Gefangenen das Recht, diese auch zu öffnen bzw. zusammenzupacken.

Diese Informationsschutzbedingungen sind aus der Darstellung von Fig. 6/1 nicht ersichtlich.

^{*)} Dies geschieht durch die Übertragung der Zugriffsrechte GET.MAILBOX bzw. GET.MAINBOX.

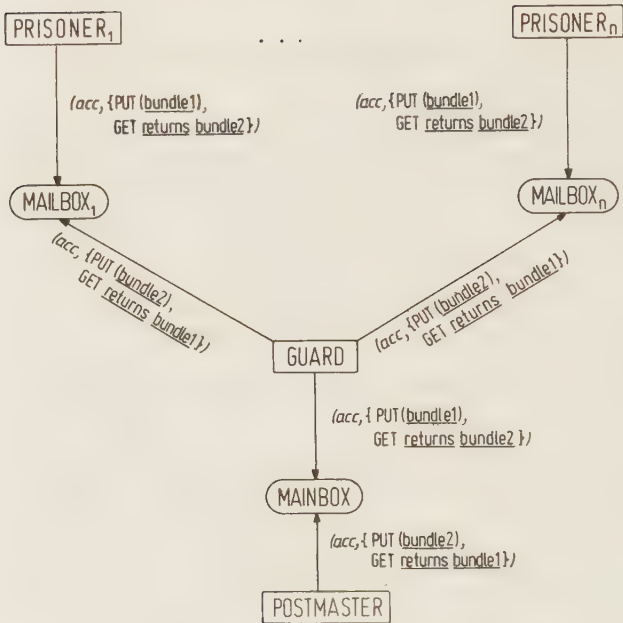


Fig. 6/1

Zur Lösung des Problems bieten sich folgende zwei Möglichkeiten an:

1. Die Objekte werden "etikettiert":

An jeder Zugriffsoperation eines Objektes (D-Moduls bzw. des zugrunde liegenden abstrakten Datentyps) wird notiert, welche Subjekte (welcher F-Modul) die Operation ausführen dürfen.*)

*) Dieser Ansatz wurde bei der Programmiersprache GIPSY (vgl. [AMH 77]) generell für alle Programmeinheiten gewählt.

In unserem Beispiel könnte dies für den Typ bundle1 (analog für bundle2) folgendermaßen aussehen:

```
type module BUNDLE1 ...

  interface

    public procedures

      $ PRISONER, POSTMASTER $ REMOVE_LETTER ...
      $ PRISONER $ WRITE_LABEL ...,
      $ PRISONER $ APPEND_LETTER ...

    end interface;

.
.

end type module BUNDLE1;
```

In Erweiterung der Syntax für "type module" aus Kap. 3.2 ist hier (in \$ eingeschlossen) für die Zugriffsoptionen REMOVE_LETTER (≙ Öffnen des Bündels), WRITE_LABEL (≙ Adressierung) bzw. APPEND_LETTER (≙ Zusammenpacken) angegeben, wer sie ausführen darf.

Diese Lösung hat den Nachteil, daß dadurch die Objekte der Art bundle1 (bzw. im allgemeinen alle D-Moduln) kontextabhängig werden, was im Widerspruch zu den Forderungen an Programmoduln aus Kapitel 1 steht.

2. Durch eine Präzisierung der E/A-Spezifikation der den übertragenen Zugriffsrechten zugrunde liegenden Operationen kann ebenfalls die in den Briefbündeln enthaltene Information geschützt werden. Wir gehen davon aus, daß bundle1 (bzw. bundle2) durch folgenden abstrakten Datentyp beschrieben wird:

```

type module BUNDLE1 ... so bzw. BUNDLE2 oc
  interface
    public procedures
      WRITE_LABEL ...
      REMOVE_LETTER ...
      APPEND_LETTER ...
    end interface;
    .
    .
    .
end type module BUNDLE1;

```

Die Spezifikation für den Wächter (F-Modul GUARD) kann nun wie folgt präzisiert werden:

GUARD

```

access to
  (MAILBOX1, ..., MAILBOXn)
    with PUT (bundle2 with none),
      GET returns (bundle1 with none);
  MAINBOX
    with PUT (bundle1 with none),
      GET returns (bundle2 with none)
  end access;

```

Die Schreibweise "GET returns (bundle1 with none)" soll anzeigen, daß auf Objekten, die als Ergebnis durch die Ausführung der Operation GET verfügbar werden, keine Zugriffsoperationen (none) durch den F-Modul GUARD ausführbar sind.

Die Spezifikation für den Postmeister kann wie folgt sein:

POSTMASTER

access to

 MAINBOX with

 PUT (bundle2 with WRITE_LABEL,
 REMOVE_LETTER,
 APPEND-LETTER) ,

 GET returns (bundle1 with REMOVE_LETTER)

end access;

□

Durch einen Vergleich der Spezifikationen für den Postmeister und den Wächter kann man zusammen mit der in Fig. 6/1 dargestellten Modul-Struktur folgendes erkennen:

Die Zugriffsberechtigung bzgl. eines Objektes der Art bundle1 (Bzw. bundle2) kann sich dynamisch ändern; je nachdem *wo* es verfügbar ist.

Solange sich der Wächter im Besitz eines Briefbündels befindet, kann dieses nicht verändert bzw. aus diesem keine Information entnommen werden. Der Postmeister hingegen hat die Möglichkeit, aus demselben Bündel (mit Hilfe der Operation REMOVE_LETTER) Briefe zu entnehmen. Um den Postmeister seinerseits daran zu hindern, die entnommenen Briefe auch zu lesen (d.h. sie zu öffnen), bedarf es einer weitergehenden Spezifikation der Operation REMOVE_LETTER, etwa in folgender Form:

POSTMASTER

access to

MAINBOX with

·
·

GET returns (bundle! with REMOVE LETTER

returns (letter with none));

end access;

Welche Klasse von Informationsschutzproblemen sich mit Hilfe der soeben vorgestellten *mehrstufigen E/A-Spezifikation* von Zugriffsoperationen darstellen läßt, soll nicht weiter Gegenstand dieser Arbeit sein. Man kann jedoch feststellen, daß die mehrstufige Spezifikation gegenüber dem "Etikettieren von Objekten" den Vorteil hat, daß die Programmoduln kontextunabhängig formulierbar sind.

KAPITEL 7 VERGLEICHE MIT ANSÄTZEN IN DER LITERATUR

Dieses abschließende Kapitel verfolgt zwei Ziele: zum einen soll eine Übersicht über einige in der Literatur bekannte Ansätze im Zusammenhang mit Datenstrukturen und modularer Programmierung gegeben, zum anderen ein Eindruck vermittelt werden, wie die vorliegende Arbeit anderen gegenüber einzuordnen ist.

Die Ansätze werden dabei nach den folgenden drei Gesichtspunkten geordnet betrachtet: formale Spezifikationsmethoden für Datenstrukturen, Systementwurfsüberlegungen, programmiersprachliche Konstruktionen für Moduln.

Wenden wir uns zunächst dem ersten Punkt zu:

Der älteste Ansatz zur Spezifikation von Datenstrukturen stammt von Earley [EAR 71]. Dort werden Datenstrukturen durch gerichtete Graphen (sogenannte V-Graphen) und die zugehörigen Operationen durch Transformationsregeln spezifiziert.

Hoare schlägt in [HOA 72] vor, die Korrektheit der Repräsentierung von Datenstrukturen bzw. deren Semantik wie folgt zu beschreiben:

Ausgehend vom Klassen-Konzept bei SIMULA 67 [DMN 70] werden unter Verwendung der Datenstrukturen im Klassen-Rumpf Zusicherungen über die Klassen-Operationen angegeben und dadurch das E/A-Verhalten derselben in bezug auf die Datenrepräsentierung charakterisiert.

Eine Weiterführung dieser ersten beiden Ideen findet man bei [WEB 78].

Bei Gutttag [GUT 75] ist eine Spezifikationsmethode für abstrakte Datentypen angegeben, die unabhängig von einer speziellen Repräsentierung der zugrunde liegenden Datenstruktur ist. Die Spezifikation soll umgekehrt als Implementierungsvorschrift für die Datenstruktur dienen. Grundlage der Methode sind heterogene Algebren und Gleichungen,

welche zur Charakterisierung von Beziehungen der Datentyp-Operationen dienen.

Die Beschreibung von zulässigen Aufrufsequenzen der Operationen einer Datenstruktur durch sogenannte "Pfad-Ausdrücke" sind Gegenstand der Spezifikationsmethode von Flon/Habermann [FLH 76]. Dieser Gedanke ist besonders im Zusammenhang mit Betriebssystemen und Synchronisationsproblemen von Interesse.

Das von uns vorgestellte Modul-Konzept beinhaltet keine Spezifikation von Datenstrukturen. Es wird dagegen ein Ansatz zur Darstellung der Vernetzungen innerhalb eines modularen Programmsystems (Modulgeflechts) angegeben. Die Methode von Hoare scheidet für unsere Zwecke aus, da in diesem Fall zur Spezifikation von D-Moduln bzw. abstrakten Datentypen Information über die Repräsentierung benötigt wird, was dem Prinzip "information hiding" widerspricht. Die Möglichkeit der Beschreibung von D-Moduln mit Hilfe von (algebraischen) Gleichungen für die Zugriffsoperationen wird verhindert durch das "Abmagerungskonzept" bzgl. der Operationen eines D-Moduls bzw. durch die Tatsache, daß bei der Übertragung von Zugriffsrechten an Moduln die Zugriffsrechte bzgl. eines Objektes eingeschränkt werden können. Die Gleichungen würden für derart abgemergerte Datenstrukturen ihre Gültigkeit verlieren.

Im Zusammenhang mit Systementwurfsüberlegungen wird gegenwärtig viel diskutiert über die "Michael Jackson-Methode" (vgl. [JAC 76]) und die von IBM entwickelte Methode HIPO (hierarchy plus input-process-output, vgl. [STA 76]).

Ausgangspunkt des ersten Ansatzes ist eine hierarchische Darstellung der drei Grundstrukturen "Sequenz, Iteration, Fallunterscheidung" der Strukturierten Programmierung. Die Tatsache, daß sehr häufig Datenstrukturen ebenfalls hierarchisch aufgebaut sind, ist Grundlage der Michael Jackson-Methode:

- Man finde, getrennt für die Ein- bzw. Ausgabe eines Systems, die hierarchische Struktur der Daten.
- Die Programmstruktur ergibt sich, indem man korrespondierende Komponenten der Ein- bzw. Ausgabedatenstruktur zusammenfaßt und jeder der Datenkomponenten ein ausführbares Verarbeitungsprogramm zuordnet.

Probleme ergeben sich hier, wenn die zu bearbeitenden Daten keine hierarchische Struktur aufweisen oder wenn zwischen den Ein- und Ausgabedaten keine direkten Beziehungen bestehen. Als nachteilig kann auch empfunden werden, daß die Verarbeitung aller beteiligten Daten über die gesamte Programmstruktur "verschmiert" ist und sich somit kleine Änderungen an den Datenstrukturen sehr stark auswirken können. Bei der Methode HIPO geht man von einer hierarchischen (baumartigen) Zerlegung eines Systems in Teilfunktionen aus, deren E/A-Verhalten mit Hilfe eines sogenannten "input-process-output"-Diagramms beschrieben wird. Zwischen den einzelnen Funktionen werden beliebige Datenbezüge zugelassen, die Methoden der Datenabstraktion bleiben unberücksichtigt. Legt man das in unserem Ansatz vorgestellte Modul-Konzept mit den entsprechend eingeschränkten Datenbezügen (Zugriffsrechten!) zugrunde, so eignet sich die Methode HIPO ebenso als Dokumentationshilfe eines modularen Programmsystems wie die von uns in Kapitel 2.3 entworfene Darstellung.

Zum Schluß soll noch auf die Beschreibung von Moduln in den Programmiersprachen MODULA [WIR 77], CDL2 ([HAE 76], [KOS 76]), CLU [LIZ 74] und ALPHARD [WUL 74] eingegangen werden.

MODULA geht von einer parallelen Programmierumgebung aus. Das von PASCAL übernommene Block-Konzept wird durch ein Modul-Konzept ergänzt. Ein Modul ist hier eine Sammlung von Konstanten-, Typ-, Variablen- bzw. Prozedurdeklaratio-

nen. Die Schnittstelle der Moduln nach außen besteht aus einer "define"-Liste und einer "use"-Liste. Die erste beschreibt die von einem Modul nach außen zur Verfügung gestellten Objekte, die zweite legt fest, welche Objekte an einen Modul von außen übertragen werden. Auf diese Weise können über Modulgrenzen hinweg beliebige Daten und Prozeduren gereicht werden, im Gegensatz zu unserem Ansatz. In MODULA werden nun drei Typen von Moduln verwendet:

- einfache Moduln (Bezeichnung: module), die - bis auf unterschiedliche Möglichkeiten der Schnittstellenspezifikation (siehe oben) - im wesentlichen mit den D-Moduln unseres Ansatzes übereinstimmen.
- Moduln zur Steuerung konkurrierender Prozesse (Bezeichnung: interface module), die im wesentlichen mit den Monitoren von Hoare und Hansen (vgl. [HOA 74], [HAN 75]) übereinstimmen.
- Moduln zur Definition logischer Geräte (Bezeichnung: device module), die als Schnittstelle zur Hardware dienen.

Programme in CDL2 bestehen aus hierarchisch geordneten Schichten (Bezeichnung: layer). Eine Schicht besteht aus einer Sammlung von Moduln (Bezeichnung: section). Ein Modul in CDL2 entspricht dabei einem D-Modul in unserer Terminologie, d.h. er besteht aus (Zugriffs-)Algorithmen und zugehörigen Daten. Über die Schnittstelle von Schichten können streng hierarchisch "von unten nach oben" Algorithmen und Konstanten gereicht werden. Diesen Vorgang nennt man in CDL2 "Abstraktion". Eine Schicht erscheint somit nach außen wie ein D-Modul, dessen Zugriffsoperationen eine "vom Programmierer zusammengestellte" Auswahl von Algorithmen der Moduln innerhalb der Schicht ist. Auf die gleiche Weise können Moduln einer Schicht miteinander kommunizieren, was dann als "Extension"

bezeichnet wird. Die Übernahme von Algorithmen aus einem Modul wird als "Invokation" bezeichnet.

Das cluster-Konzept von CLU diene als Vorlage für unser D-Modul-Konzept. Die Erweiterung besteht zum einen darin, daß an D-Moduln - über die Schnittstelle nach außen - Zugriffsrechte übertragen werden können, zum anderen kann man innerhalb von D-Moduln weitere F- bzw. D-Moduln definieren.

Die Programmiersprache ALPHARD bietet ein interessantes Konzept zur Spezifikation abstrakter Datentypen, die als form bezeichnet werden und in unserem Konzept einem Typ-Modul (type module) entsprechen. Eine form besteht aus den Komponenten "Spezifikation", "Repräsentierung" und "Implementierung". Mit Hilfe einer "logischen Sicht" der Repräsentierung werden im Spezifikationsteil Eigenschaften der zugrunde liegenden Datenstruktur angegeben. Das dynamische Verhalten der Daten wird durch Zusicherungen bzgl. der Zugriffsoperationen spezifiziert.

SCHLUSSBEMERKUNG

An dieser Stelle möchte ich mich bei Herrn Prof. Dr. H. J. Schneider für die Betreuung der Arbeit und Herrn Prof. Dr. F. Hofmann für die Übernahme des Zweitgutachtens bedanken.

Mein Dank gilt auch meinen Kollegen H. Göttler, M. Nagl, G. Stöth und D. Weber vom Lehrstuhl für Programmiersprachen und H. Bunke vom Lehrstuhl für Mustererkennung, die durch ihre Bereitschaft zur Diskussion und durch Hinweise zu dieser Arbeit beigetragen haben.

Zu danken ist auch Frau J. Ott-Lexa für die Unterstützung beim Austesten von Programmbeispielen, Herrn A. Wanninger für die exakte zeichnerische Ausarbeitung graphischer Darstellungen und Frau E. Lührs für die sorgfältige Erstellung des Manuskriptes.

STICHWORTVERZEICHNIS

- abstrakter Datentyp 74
- Abstraktion 1
 - datenorientierte ~ 1
 - funktionale ~ 1
 - syntaktische ~ 2
- Anschlußpunkte 52
- Art, eines D-Moduls 74
- art(..), Art einer Kante zwischen (Modul-)Knoten eines → Modulgraphen 30
- Datenmodul, D-Modul 8, 46, 74
- D-Modul-Deklaration 79, 87
- D-Modul-Implementierung 79, 86
- D-Modul-Struktur 33
- D-Modul-Zugriff 80, 89
- datenorientiertes System 11, 42
- Datentyp-Deklaration 77, 84
- Datentyp-Implementierung 77, 82
- Definitionskanten 53
- Entwurfsentscheidungen, Reihenfolge von ~ 14
- Fehlerausgang 18, 89
- funktionaler Modul, F-Modul 8, 47, 75
- F-Modul-Aufruf 81
- F-Modul-Aufrufverschachtelung 30
- F-Modul-Implementierung 80, 90
- F-Modul-Struktur 30
- funktionsorientiertes System 11, 42
- Hierarchie-Verträglichkeit
 - bzgl. D-Moduln 35
 - bzgl. F-Moduln 34
- information hiding 10, 36
- information security 10
- Informationsverteilung, Kontrolle der ~ 15
- K, Knotenmenge 24
- kamark(..), Kantenmarkierungsfunktion 25
- knomark(..), Knotenmarkierungsfunktion 25
- Liste BESCHROP, Beschreibung der Operationen 114
- Liste BESCHRÜZR, Darstellung der an die Moduln übertragenen Zugriffsrechte 117
- Liste DMOBESCHR, Beschreibung der D-Moduln 110
- Liste FMOBESCHR, Beschreibung der F-Moduln 107
- Liste PARBESCHR, Parameterbeschreibungen 115
- mehrstellige Operationen 160
- Modul 7
- modular 7
- modulare Programmierung 8
- modulares Programmsystem 9
- Modulgeflecht 44
- Modulgeflecht, strukturelles ~ 58
- Modulgraph 24
- Modulgraph, konsistenter ~ 44
- Parametrisierung 49
- Realisierung von D-Moduln
 - in ALGOL 60 142, 143
 - in FORTRAN 144, 145
 - in TAS 147, 149, 151
- Realisierung von F-Moduln
 - in ALGOL 60 153, 155
 - in FORTRAN 154, 156
 - in TAS 154, 157
- Ressourcen, zur Verfügung stehende ~ 51
- Ressourcen-Verwaltung, zentrale ~ 10, 40
- r_{FF}' , r_{FD}' , r_{DD}' , r_{DF} 25
- R, Kantenrelation 24
- Schnittstelle
 - von außen übertragene ~ eines D-Moduls 12, 51
 - eines D-Moduls nach innen 13, 51
 - eines F-Moduls nach außen 11, 51
 - eines F-Moduls nach innen 11
 - nach außen zur Verfügung gestellte ~ eines D-Moduls 12, 51

Schutzsystem

- Objekt-~ 163

- Informations-~ 163

Transparenz 17

Untermodulekanten 53

Untermodule-Spezifikation 80, 92

übertragene Zugriffsrechte 45

virtuelle Maschine 17

Zerlegung, Kriterien der ~ 16

Zerlegungsrichtung 16

zugriffe(..), Zugriffsoperationen,
die zur Markierung
einer + Zugriffs-
oder + Definitions-
kante dienen 30

Zugriffskante 53

Zugriffsoperation 8

Zugriffsrecht 9, 51

Zugriffsrechte, Verteilung der ~ 10

LITERATURVERZEICHNIS

- [ALT 74] *Altmann, W.*: Konstruktionen zur Erweiterbarkeit von Sprachen, Arbeitsber. d. Inst. f. Math. Masch. u. DV, Bd. 7, Nr. 6, Erlangen, Nov. 1974
- [ALW 77] *Altmann, W./Weber, D.*: Programmiermethodik, Arbeitsber. d. Inst. f. Math. Masch. u. DV, Bd. 10, Nr. 3, Erlangen, März 1977
- [AMH 77] *Ambler, A.L./Hoch, C.G.*: A Study of Protection in Programming Languages, SIGPLAN Notices, Vol. 12, Nr. 3, März 1977, S. 25 - 40
- [BAK 72] *Baker, F.T.*: Chief Programmer Team Management of Production Programming, IBM Systems Journal, Vol. 11, Nr. 1, 1972, S. 56 - 73
- [BAM 73] *Baker, F.T./Mills, H.D.*: Chief Programmer Teams, Datamation, Vol. 19, Nr. 12, Dez. 1973, S. 58 - 61
- [DDH 72] *Dahl, O.J./Dijkstra, E.W./Hoare, C.A.R.*: Structured Programming, Academic Press, London, 1972
- [DEK 76] *DeRemer, F.L./Kron, H.*: Programming-in-the-Large versus Programming-in-the-Small, IFB 1, 1976, S. 80 - 89
- [DEN 72] *Dennis, J.B.*: Modularity, Lecture Notes in Economics and Mathematical Systems, F.L. Bauer (Ed.), Bd. 81, 1972, S. 128 - 182
- [DEN 75] *Dennis, J.B.*: An Example of Programming with Abstract Data Types, SIGPLAN Notices, Vol. 10, Nr. 7, Juli 1975, S. 25 - 29
- [DEN 76] *Denning, P.J.*: A hard look at structured programming, Structured Programming, D. Bates (Ed.), Infotech International Ltd., Maidenhead, 1976, S. 183 - 202
- [DER 74] *DeRemer, F.L.*: A conceptual frame work for mode definitions in programming languages, IFIP M.O.L. Bulletin Nr. 4, Mai 1974, S. 15 - 31
- [DIC 71] *Dickmann, B.N.*: ETC - an extensible macro-based compiler, Proc. AFIPS, 1971 SJCC, S. 529 - 538
- [DIJ 68] *Dijkstra, E.W.*: The Structure of the T.H.E. Multiprogramming System, CACM, Vol. 11, 1968, S. 341 - 346
- [DIJ 76] *Dijkstra, E.W.*: Programming Methodologies: Their Objects and Their Nature, Structured Programming, Infotech International Ltd., Maidenhead, 1976, S. 203 - 216
- [DMN 70] *Dahl, O.J./Myhrhaug, B./Nygaard, K.*: The SIMULA 67 Common Base Language, Norwegian Computing Center, Oslo, 1970
- [DOD 77] Department of Defense (USA): Requirements for High Order Computer Programming Languages (Revised 'Ironman'), Juli 1977

- [EAR 71] *Earley, J.*: Toward an understanding of data structures, CACM, Vol. 14, Nr. 10, Okt. 1971, S. 617 - 627
- [ENW 74] *Encarnação, J./Weck, G.*: Eine Implementierung von DATAS, Bericht Nr. A74-1, 1974, Inst. f. Angew. Mathematik u. Informatik, Universität d. Saarlandes
- [FLH 76] *Flon, L./Habermann, A.N.*: Towards the Construction of Verifiable Software Systems, SIGPLAN Notices, Vol. 8, Nr. 2, Febr. 1976, S. 141 - 148
- [GAL 78] *Gall, R.*: Neustrukturierung von DATAS unter Zugrundelegung einer 32-Bit Maschine, Diplomarbeit am Lehrstuhl f. Programmiersprachen, Universität Erlangen-Nürnberg, 1978
- [GEM 75] *Geschke, C.M./Mitchell, J.C.*: On the Problem of Uniform Reference to Data Structures, SIGPLAN Notices, Vol. 10, Nr. 6, Juni 1975, S. 31 - 42
- [GOO 74] *Goos, G.*: Program Modules in Programming Languages, IFIP M.O.L. Bulletin Nr. 4, Mai 1974, S. 103 - 108
- [GOO 75] *Goodenough, J.B.*: Exception Handling Design Issues, SIGPLAN Notices, Vol. 10, Nr. 7, Juli 1975, S. 41 - 45
- [GUT 75] *Gutttag, J.V.*: The Specification and Application to Programming of Abstract Data Types, TR CSRG - 59, Sept. 1975, University of Toronto
- [GÜS 72] *Güntsch, F.R./Schneider, H.J.*: Einführung in die Programmierung digitaler Rechenautomaten, Walter de Gruyter, Berlin-New York, 1972
- [HAE 76] *Haeger, H.*: Benutzeranleitung für die Programmiersprache CDL 2, Arbeitspapier der Forschungsgruppe Programmiersprachen und Compiler II, TU Berlin, Juni 1976
- [HAN 75] *Hansen, P.B.*: The programming language CONCURRENT PASCAL, Proc. International Summer School on Language Hierarchies and Interfaces, München 1975
- [HOA 72] *Hoare, C.A.R.*: Proof of correctness of data representation, Acta Informatica 1, 1972, S. 271 - 281
- [HOA 74] *Hoare, C.A.R.*: Monitors: An operating system structuring concept, CACM, Vol. 17, Nr. 10, Okt. 1974, S. 549 - 557
- [HOL 77] *Holager, P.*: Some program structuring rules applied to an example, M.O.L. Bulletin Nr. 6, Febr. 1977, S. 49 - 84
- [HOR 76] *Horning, J.J.*: Some Desirable Properties of Data Abstraction Facilities, SIGPLAN Notices, Vol. 7, Nr. 2, 1976, S. 60 - 63
- [ICF 77] *Ichbiah, J.D./Ferran, G.*: Separate definition and compilation in LIS and its implementation, M.O.L. Bulletin Nr. 6, Febr. 1977, S. 17 - 26
- [JAC 76] *Jackson, M.A.*: Data structure as a basis for program design, Structured Programming, D.Bates (Ed.), Infotech International Ltd., Maidenhead, 1976, S. 279 - 291

- [JÄP 77] *Jäpel, D.*: Ein Begriffssystem zur Formulierung von Schutzaspekten und seine Anwendung auf Programmsysteme, Arbeitsber. d. Inst. f. Math. Masch. u. DV, Bd. 10, Nr. 7, Erlangen, Juni 1977
- [JOL 76] *Jones, A.K./Liskov, B.H.*: A languages extension for controlling access to shared data, IEEE Trans. Software Engineering, SE-2, 4 (Dez. 1976), S. 277 - 285
- [KNU 71] *Kruth, D.E.*: Top - Down Syntax Analysis, Acta Informatica, Vol. 1, 1971, S. 79 - 110
- [KOS 74] *Koster, C.H.A.*: Beating the global, IFIP M.O.L. Bulletin Nr. 4, Mai 1974, S. 109 - 115
- [KOS 76] *Koster, C.H.A.*: Visibility and Types, SIGPLAN Notices, Vol. 8, Nr. 2, Febr. 1976, S. 179 - 190
- [LEA 66] *Leavenworth, B.M.*: Syntax Macros and Extended Translation, CACM, Vol. 9, Nr. 11, Nov. 1966, S. 790 - 793
- [LHLM77] *Lampson, B.W./Horming, J.J./London, R.L./Mitchell, J.G./Popeh, G.J.*: Report On The Programming Language Euclid, SIGPLAN Notices, Vol. 12, Nr. 2, Febr. 1977
- [LIZ 74] *Liskov, B./Zilles, S.N.*: Programming With Abstract Data Types, SIGPLAN Notices, Vol. 9, Nr. 4, April 1974, S. 50 - 59
- [LIZ 75] *Liskov, B./Zilles, S.N.*: Specification Techniques for Data Abstraction, SIGPLAN Notices, Vol. 10, Nr. 6, Juni 1975, S. 72 - 87
- [MEI 77] *Meisner, L.P.*: FORTRAN 77, SIGPLAN Notices, Vol. 12, Nr. 1, Jan. 1977, S. 93 - 94
- [MIW 76] *Mitchell, J./Wegbreit, B.*: A Next Step in Data Structuring for Programming Languages, SIGPLAN Notices, Vol. 8, Nr. 2, Febr. 1976, S. 69 - 70
- [MOR 73] *Morris, J.H.*: Protection in Programming Languages, CACM, Vol. 16, Nr. 1, Jan. 1973, S. 213 - 229
- [PAL 73] *Palme, J.*: Protected program modules in SIMULA 67, FOA Report, C8372-M3(E5), Juli 1973
- [PAL 76] *Palme, J.*: The class concept in the SIMULA programming language, FOA Report, C10052-M3(E5), Aug. 1976
- [PAR 71] *Parnas, D.L.*: Information Distribution Aspects of Design Methodology, Proceedings IFIP Congress 1971, Ljubljana, S. 339 - 344
- [PAR 72A] *Parnas, D.L.*: A Technique for Software Module Specification with Examples, CACM, Vol. 15, Nr. 5, Mai 1972, S. 330 - 336
- [PAR 72B] *Parnas, D.L.*: On the Criteria To Be Used in Decomposing Systems into Modules, CACM, Vol. 15, Nr. 12, Dez. 1972, S. 1053 - 1058
- [PAS 75] *Parnas, D.L./Siewiorek, D.P.*: Use of Concept of Transparency in the Design of Hierarchically Structured Systems, CACM, Vol. 18, Nr. 7, Juli 1975, S. 401 - 408

- [PRO 77] Proceedings of an ACM Conference on Language Design for Reliable Software, SIGPLAN Notices, Vol. 12, Nr. 3, März 1977
- [PSE 75] Parnas, D.L./Shore, J./Elliott, D.: On the need for fewer restrictions in changing compile-time environments, SIGPLAN Notices, Vol. 10, Nr. 5, Mai 1975, S. 29 - 36
- [PSW 76] Parnas, D.L./Shore, J.E./Weiss, D.: Abstract Types Defined as Classes of Variables, SIGPLAN Notices, Vol. 8, Nr. 2, Febr. 1976, S. 149 - 154
- [REI 77] Reitz, K.-H.: Durch Systematisierung zu einem freiprojektierbaren Software-Baustein zur Lösung von Automatisierungsaufgaben im Sinne von Regeln, Steuern und Überwachen, IFB7, Fachtagung Prozessrechner, 1977, S. 109 - 120
- [SCH 70] Schneider, H.J.: Chomsky-Systeme für partielle Ordnungen, Arbeitsber. d. Inst. f. Math. Masch. u. DV, Bd. 3, Nr. 3, Erlangen, Aug. 1970
- [SCH 74] Schuman, S.A.: Toward Modular Programming in High-Level Languages, IFIP M.O.L. Bulletin Nr. 4, Mai 1974, S. 117 - 128
- [SCH 75] Scheidig, H.: Einige Überlegungen zur Modularisierung von Systemen und System-Programmiersprachen, TUM Bericht Nr. 7506, 1975
- [SCH 75a] Schneider, H.J.: Compiler - Aufbau und Wirkungsweise, De Gruyter Verlag, Berlin-New York, 1975
- [SCJ 70] Schuman, S.A./Jorrand, P.: Definition mechanisms in extensible programming languages, Proc. AFIPS, 1970 FJCC, Vol. 37, S. 9 - 20
- [SHA 76] Shaw, M.: Research Directions in Abstract Data Structures, SIGPLAN Notices, Vol. 8, Nr. 2, Febr. 1976, S. 66 - 68
- [SIM 76] SIMAT Projektierungshandbuch der Firma SIEMENS, Best.-Nr. P71100-S0031-X-A2-35
- [SOY 74] Solntseff, N./Yezerski, A.: A Survey of Extensible Programming Languages, Ann. Rev. in Aut. Programming, Vol. 7, Part. 5, 1974, S. 264 - 301
- [STA 75] Standish, T.A.: Extensibility in programming language design, Proc. National Computer Conference 1975, S. 287 - 290
- [STA 76] Stay, J.F.: HIPO and integrated program design, IBM Systems Journal, Vol. 15, Nr. 2, 1976, S. 143 - 154
- [WEB 78] Weber, D.: Datengraphen und deren Transformation: Ein Konzept zur Spezifikation von Datentypen, Dissertation, Arbeitsber. d. Inst. f. Math. Masch. u. DV, Bd. 11, Nr. 8, Erlangen, Juni 1978

- [WEG 76] *Wegner, P.*: Programming Languages - The First 25 Years, IEEE Transactions on Computer, Vol. C-25, Nr. 12, Dez. 1976
- [WIJ 69] *van Wijngaarden, A. et. al.*: Report on the Algorithmic Language ALGOL 68, Num. Math., 14, 1969, S. 79 - 218
- [WIR 71] *Wirth, N.*: The programming language PASCAL, Acta Informatica, Vol. 1, 1971, S. 35 - 63
- [WIR 77] *Wirth, N.*: Modula: A Language for Modular Multiprogramming, Software - Practice and Experience. Vol. 7, 1977, S. 3 - 35
- [WLS 76] *Wulf, W.A./London, R.L./Shaw, M.*: An Introduction to the Construction and Verification of Alphard Programs, IEEE Transactions in Software Engineering, Vol. SE-2, Nr. 4, Dez. 1976, S. 253 - 264
- [WRH 71] *Wulf, W.A./Russell, B.B./Habermann, A.N.*: BLISS: A Language for System Programming, CACM, Vol. 14, Nr. 12, Dez. 1971, S. 780 - 790
- [WUL 74] *Wulf, W.A.*: ALPHARD: A Language to support Structured Programs, Carnegie-Mellon University, 1974
- [WUS 73] *Wulf, W./Shaw, M.*: Global variables considered harmful, SIGPLAN Notices, Vol. 8, Nr. 2, Feb 1973, S. 28 - 33
- [ZIL 73] *Zilles, S.N.*: Procedural encapsulation: a linguistic protection technique, SIGPLAN Notices, Vol. 8, Nr. 9, Sept. 1973, S. 140 - 146

LEBENS LAUF

geboren: 23. April 1950 in Fürth/Bay., als
Sohn des städtischen Angestellten
Josef Altmann und seiner Frau Johanna
geborene Ortner

1956 - 1960: Besuch der Volksschule Oedenberger-
straße in Nürnberg

1960 - 1969: Besuch des Hans-Sachs-Gymnasiums in
Nürnberg

28. 6. 1969: Erlangung der Hochschulreife

1969 - 1974: Studium der Mathematik bzw. der Informatik
an der Universität Erlangen-Nürnberg

21. 1. 1974: Erlangung des akademischen Grades
"Diplom-Informatiker" an der Universität
Erlangen-Nürnberg

derzeitige Tätigkeit: Verwalter der Dienstgeschäfte eines
wissenschaftlichen Assistenten am
Lehrstuhl für Programmiersprachen der
Universität Erlangen-Nürnberg

derzeitiger Wohnsitz: Oedenbergerstr. 160, 8500 Nürnberg

In letzter Zeit sind folgende Arbeitsberichte des Instituts
für Mathematische Maschinen und Datenverarbeitung erschienen :
=====

Band 9
=====

- NR. 1 Brendel, W., FrUhauf, T., Nagl, M. :
Graph Rewriting Systems II.
Three Papers on Theory, Application, and Implementation.
(Februar 1976)
- NR. 2 Brügmann, H. W., Rudert, A., Lehr, H. :
EBIS
Strukturierte Bildprogrammierung auf höherer Ebene.
(Februar 1976).
- NR. 3 Hofmann, W., Wichmann, H. :
Konzept der Vertikalverarbeitung.
Implementierung und Analyse vertikaler Algorithmen
auf einem mikroprogrammierten Rechner.
(Mai 1976)
- NR. 4 Jordan, H. F., Smith, B. J. :
The Hardware-Software Interface.
(Juni 1976)
- NR. 5 Gernoth, B., Schmidt, B. :
Das Modell MEM-SIM.
Beschreibung eines Simulationsmodells zur Unter-
suchung einer Rechenanlage mit einer symmetrischen
Mehrprozessorkonfiguration und privaten Speichern.
(Oktober 1976)
- NR. 6 Grötsch, E. :
Vergleichende Studie über parallele Ersetzung
markierter Graphen.
Jahn, B. :
Eine Studie zur Implementierung von Graph-Grammatiken
unter Verwendung von Assoziativverarbeitung.
(Dezember 1976)
- NR. 7 Mertens, P., Neuwirth, W., Purucker, G.,
Schaller, K., Schmitt, W. :
Verknüpfung von Daten- und Methodenbanken, dargestellt
am Beispiel der Analyse von Marktforschungsdaten.
(Dezember 1976)
- NR. 8 Parallelismus in der Informatik.
Wissenschaftliches Kolloquium aus Anlass des Zehnjährigen
Bestehens des IMMD, Erlangen, 10./11. Juni 1976.
(Dezember 1976)
- NR. 9 Jahresbericht 1976 der Informatik.
(März 1977)

Band 10
=====

- NR. 1 Hoffmann, W. :
Modelle und Analysen von Teilnehmerrechensystemen,
angewendet auf das Teilnehmersystems RESPOND unter
CD 3300 MASTER.
(Februar 1977)
- NR. 2 Körber, K., Werzinger, G. :
Das Modell SIM-LAB.
Simulation eines klinisch-chemischen Laboratoriums.
(März 1977)
- NR. 3 Altmann, W., Weber, D. :
Programmiermethodik.
(März 1977)
- NR. 4 Maehle, E. :
Fehlertolerante Rechnerstrukturen.
(April 1977)
- NR. 5 Beth, T. :
Kombinatorik: Theorie der Vertretersysteme.
(April 1977)
- NR. 6 Mertens, P. :
Einige Hinweise zur technischen Gestaltung von
Bildschirmdialogen.
(Mai 1977)
- NR. 7 Jäpel, D. :
Ein Begriffssystem zur Formulierung von Schutzaspekten
und seine Anwendung auf Programmsysteme.
(Juni 1977)
- NR. 8 Brendel, W., Grötsch, E., Nagl, M., Schneider, H.-J. :
Graphersetzungssysteme, Beiträge zur Theorie, Anwen-
dung und Implementierung.
(Juni 1977)
- NR. 9 Rüdel, W. :
Darstellung und Vergleich abstrakter Modelle zur
Beschreibung geschützter Systeme.
(Juli 1977)
- NR. 10 Bolch, G., Gall, R., Grassl, M., Schmidt, B. :
Modelle für Trommelspeicher.
(Juli 1977)
- NR. 11 v. Eichhorn, K. :
Steiner Systeme, Codes und affine Räume.
(Juli 1977)
- NR. 12 Göttler, H. :
Zweistufige Graphmanipulationssysteme für die
Semantik von Programmiersprachen.
(Juli 1977)

Band 10 (Forts.)

=====

- NR. 13 Keramidis, S. :
Ein allgemeines formales Modell für Betriebssysteme und dessen Implementierung.
(September 1977)
- NR. 14 Arnold, F., Benker, L., Grütz, M. :
Die Fast Fourier Transformation und Möglichkeiten ihrer Implementierung auf EGPA.
(November 1977)
- NR. 15 Gardill, R., Händler, W., Hessling, H.,
Klar, R., Spies, P. P. :
ERES
Eine nichtprozedurale Rechnerentwurfssprache mit präziser Zeitbeschreibung.
(November 1977)
- NR. 16 Wichmann, H. :
Entwicklung und Bewertung von Algorithmen für ein mikroprogrammiertes Parallel- und Assoziativkonzept auf Mikroprogramm- und Benutzerebene.
(Dezember 1977)
- NR. 17 Brüggmann, H. W. :
EBIDIS - Strukturierte Bildprogrammierung im Dialog.
(Dezember 1977)
- NR. 18 Jahresbericht 1977 der Informatik.
(März 1978)

Band 11

=====

- NR. 1 Körber, K., Werzinger, G. :
Das Modell SIM-LAB.
Beschreibung des Grundmodells.
(Februar 1978)
- NR. 2 Körber, K., Werzinger, G. :
Das Modell SIM-LAB.
Beschreibung der Modellerweiterungen und der Strategien.
(Februar 1978)
- NR. 3 Körber, K., Werzinger, G. :
Das Modell SIM-LAB.
Listing der Quellprogramme.
(Februar 1978)
- NR. 4 Beth, Th. :
On Resolutions of Steiner Systems.
(März 1978)
- NR. 5 Leeb, K. :
Salami-Taktik beim Quader-Packen.
Riess, W. :
Zwei Optimierungsprobleme auf Ordnungen.
(April 1978)
- NR. 6 Händler, W., Henning, W., Kleinöder, W., Kneissl, J.,
Volkert, J. :
EPIK - Erlanger Projekt Interaktive Kartographie.
(April 1978)
- NR. 7 Schreiber, H. :
Hardware-Messung und Analyse des Ablaufgeschehens
in Rechnerkernen.
(April 1978)
- NR. 8 Weber, D. :
Datengraphen und deren Transformation :
Ein Konzept zur Spezifikation von Datentypen.
(Juni 1978)
- NR. 9 Musielak, H., Schmidt, B., Stössel, M. :
Vergleich von Simulationssprachen :
GPSS-FORTRAN - GPSS - GASP IV.
(Juli 1978)
- NR. 10 Solymosi, A. :
Synthese von analysierenden Automaten auf Grund
von formalen Grammatiken.
(Juli 1978)
- NR. 11 Meyer, K., Schaller, K., Schmidt, B. :
SIM-QUEUE.
Beschreibung eines Simulationsmodells für ein
komplexes Warteschlangensystem, das mit Hilfe von
Eingabedaten zusammengestellt werden kann.
(Juli 1978)

Band 11 (Forts.)

=====

- NR. 12 Bodendorf, F., Eckardt, T., Mertens, P., Schrammel, D. :
Benutzung und Beurteilung von Kleinrechnersystemen in
einer Hochschule - Ein Beitrag zur Dezentralisierungs-
diskussion.
(August 1978)
- NR. 13 Rabinovitz, M. :
An Approach to a Reconfigurable Multiprocessor System.
Bode, A., Händler, W. :
Classification d'architectures parallèles:
Introduction de la notation ECS et application au
projet EGPA.
Sigmund, W. :
Parallel Machine Functions in Program Transformation
and Execution.
(September 1978)
- NR. 14 Beth, Th., Strehl, V. :
Materialien zur Codierungstheorie.
(September 1978)
- NR. 15 Wendler, K. :
Betriebssystemaspekte in hierarchisch modularen
Polyprozessorsystemen - Modellierungsansätze und
Koordinierungsmechanismen.
(Oktober 1978)
-

Offsetdruck-Fotodruck

J. Hogl, 8520 Erlangen, Hauptstr. 109 (Nähe Martin-Luther-Platz) Tel. 22990